The Combined KEAPPA - IWIL Workshops Proceedings

Proceedings of the workshops

Knowledge Exchange: Automated Provers and Proof Assistants

and

The 7th International Workshop on the Implementation of Logics

held at

The 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning
November 23-27, 2008, Doha, Qatar

Knowledge Exchange: Automated Provers and Proof Assistants (KEAPPA)

Existing automated provers and proof assistants are complementary, to the point that their cooperative integration would benefit all efforts in automating reasoning. Indeed, a number of specialized tools incorporating such integration have been built. The issue is, however, wider, as we can envisage cooperation among various automated provers as well as among various proof assistants. This workshop brings together practitioners and researchers who have experimented with knowledge exchange among tools supporting automated reasoning.

Organizers: Piotr Rudnicki, Geoff Sutcliffe

The 7th International Workshop on the Implementation of Logics (IWIL)

IWIL has been unusually sucessful in bringing together many talented developers, and thus in sharing information about successful implementation techniques for automated reasoning systems and similar programs. The workshop includes contributions describing implementation techniques for and implementations of automated reasoning programs, theorem provers for various logics, logic programming systems, and related technologies.

Organizers: Boris Konev, Renate Schmidt, Stephan Schulz

Automated Reasoning for Mizar: Artificial Intelligence through Knowledge Exchange

Josef Urban*
Charles University in Prague

Abstract

This paper gives an overview of the existing link between the Mizar project for formalization of mathematics and Automated Reasoning tools (mainly the Automated Theorem Provers (ATPs)). It explains the motivation for this work, gives an overview of the translation method, discusses the projects and works that are based on it, and possible future projects and directions.

1 Introduction and Motivation

1.1 Why Mizar?

The Mizar proof assistant [Rud92, RT99] was chosen by the author for experiments with automated reasoning tools because of its focus on building the large formal Mizar Mathematical Library (MML) [RT03]. This formalization effort was started in 1989 by the Mizar team, and its main purpose is to verify a large body of mainstream mathematics in a way that is close and easily understandable to mathematicians, allowing them to build on this library with proofs from more and more advanced mathematical fields. This formalization goal has influenced:

- the choice of a relatively human-oriented formal language in Mizar
- the choice of the declarative Mizar proof style (Jaskowski-style natural deduction)
- the choice of first-order logic and set theory as unified common foundations for the whole library
- the focus on developing and using just one human-obvious [Rud87] first-order justification rule in Mizar
- and the focus on making the large library interconnected, usable for more advanced formalizations, and using consistent notation.

There are other systems and projects that are similar to Mizar in some of the above mentioned aspects. For example, building large and advanced formal libraries seems to be more and more common today, probably also because of the recent large formalization projects like Flyspeck [CLR06] that require a number of previously proved nontrivial mathematical results. In the work that is described here, Mizar thus should be considered as a suitable particular choice of a system for formalization of mathematics, which uses relatively common and accessible foundations, and produces a large formal library written in a relatively simple and easy-to-understand style. Some of the author's systems described below also work with other than Mizar data: for example, the learning from proofs has been experimentally instantiated to learn from Hales' proof of the Jordan Curve Theorem done in HOL Light[1] [HSA06], while MaLARea has already been successfully used for reasoning over problems from the large formal SUMO ontology[2] [NP01].

Rudnicki P, Sutcliffe G., Konev B., Schmidt R., Schulz S. (eds.);
Proceedings of the Combined KEAPPA - IWIL Workshops, pp. 1-16

*A large part of the work described here was supported by a Marie Curie International Fellowship within the 6^{th} European Community Framework Programme.

[1] `http://lipa.ms.mff.cuni.cz/~urban/holpademo.html`
[2] `http://www.ontologyportal.org/reasoning.html`

1.2 Benefits of cooperation

There are three kinds of benefits in linking proof assistants like Mizar and their libraries with Automated Reasoning technology and particularly ATPs:

- The obvious benefits for the proof assistants and their libraries. Automated Reasoning can provide a number of tools and strong methods that can assist formalization.
- The (a bit less obvious) benefits for the field of Automated Reasoning. For example, research in automated reasoning over very large libraries is painfully theoretical (and practically useless) until such libraries are really available for experiments. Mathematicians (and scientists, and other human "reasoners") typically know a lot of things about the domains of discourse, and use the knowledge in many ways that include many heuristical methods. It thus seems unrealistic (and limiting) to develop the automated reasoning tools solely for problems that contain only a few axioms, make little use of previously accumulated knowledge, and do not attempt to further accumulate and organize the body of knowledge.
- The benefits for the field of general Artificial Intelligence. These benefits are perhaps the least mentioned ones[3], however to the author they appear to be the strongest long-term motivation for this kind of work. In short, the AI fields of *deductive reasoning* and the *inductive reasoning* (represented by machine learning, data mining, knowledge discovery in databases, etc.) have so far benefited relatively little from each other's progress. This is an obvious deficiency in comparison with the human mind, which can both inductively suggest new ideas and problem solutions based on analogy, memory, statistical evidence, etc., and also confirm, adjust, and even significantly modify these ideas and problem solutions by deductive reasoning and explanation, based on the understanding of the world. Repositories of "human thought" that are both large (and thus allow the inductive methods), and have precise and deep semantics (and thus allow deduction) should be a very useful component for cross-fertilization of these two fields. Large formal mathematical libraries are currently the closest approximation to such a computer-understandable repository of "human thought" usable for these purposes. To be really usable, the libraries however again have to be presented in a form that is easy for existing automated reasoning tools to understand.

As mentioned above, Mizar is not the only system that can be used for experiments with automated reasoning and AI methods, and the work described here is related to a number of other works and projects. A lot of work on translating Mizar for automated reasoning tools was previously done in the ILF project [DW97]. The Isabelle/HOL system has been recently linked to first-order ATP systems [MP08, MP06], and this link is already widely used by the Isabelle community. The SAD proof verification system [VLP07] has been using ATP systems quite extensively. A translation of the Cyc knowledge base to first-order logic exists [RPG05], as well as a translation of the above mentioned SUMO ontology [PS07]. Machine learning and other complementary AI methods for formal mathematics have been studied, e.g., in [DFGS99, Len76, Col02, CMSM04, Faj88],

1.3 Structure of this paper

In Section 2 we give a brief overview of the Mizar extensions over pure first-order logic, like its type system and second-order constructs. Section 3 summarizes the methods that are used for translating the Mizar formalism and library to a pure first-order format suitable for automated reasoning tools. Section 4

[3] This may also be due to the frequent feeling of too many unfulfilled promises and too high expectations from the general AI, that also led to the current lack of funding for projects mentioning Artificial Intelligence.

shows several automated reasoning, proof assistance, and AI experiments and projects that are based on the current translation. Section 5 discusses some future possibilities.

2 Main Mizar extensions to pure first-order logic

2.1 Mizar types

Mizar is to a large extent a first-order system (using set-theoretical axioms), enhanced with a number of extensions that (are supposed to) make the formalization in Mizar more human-friendly. The largest of these extensions is the Mizar type system. Mizar allows defining *types* from first-order predicates. The difference between types and predicates in Mizar is not semantic: both are first-order predicates. The difference is technical and practical: representing some predicates as Mizar types allows the use of several type-related automations in Mizar. An example of this is automated use of type hierarchies, i.e., hierarchies saying that the descendant type is a subclass of the ancestor type. Table 1 shows such a hierarchy of type definitions started at the Mizar type "`Matrix of m,n,D`" in the article `MATRIX_1` [Jan91]. The exact meaning of the symbols and definitions mentioned there can be best explored in a linked presentation. A particular one (also used for linking with ATP tools) is available at `http://www.tptp.org/MizarTPTP`, where the following type hierarchy starts at `http://www.tptp.org/MizarTPTP/Articles/matrix_1.html#M1`.

These type declarations allow Mizar to automatically infer that any term with the type "`Matrix of m,n,D`" has also the following types, i.e., that the corresponding predicates hold about the term:

```
Matrix of D
tabular FinSequence of D*
FinSequence-like PartFunc of NAT,D*
Function-like Relation of NAT,D*
Subset of [:NAT,D*:]
Element of bool [:NAT,D*:]
set
```

It can be seen that these *Mizar types* consist of several parts: they can have one or more (unparameterized) *adjectives* like `tabular`, `Finsequence-Like`, and `Function-like`, and a (possibly parametrized) *type radix* like `Matrix of m,n,D`, `Matrix of D`, `FinSequence of D*`, `PartFunc of NAT,D*`, `Relation of NAT,D*`, etc. While only unidirectional widening hierarchies (like the one in Table 1) are allowed for the *type radices*, the mechanisms used for Mizar *adjectives* allow practically arbitrary Horn clauses (called *clusters* in Mizar), which results in fixpoint-like algorithms for computing the complete set of adjectives of a given term. For example, a term of the above mentioned type "`Matrix of m,n,D`" would automatically inherit all the adjectives of its type's ancestors (`tabular`, `Finsequence-like`, and `Function-like`), but it would also get all adjectives added by Mizar *clusters* to this initial set of adjectives. An example of such a cluster[4] is:

```
registration
cluster FinSequence-like -> finite set;
end;
```

This can be understood as the Horn clause:

```
finite :- FinSequence-like.
```

[4] `http://www.tptp.org/MizarTPTP/Articles/finseq_1.html#CC1`

Table 1: Hierarchy of type definitions for "Matrix of m,n,D"

```
definition
let D be non empty set;
let m, n be Nat;
mode Matrix of m,n,D -> Matrix of D means :Def3:  ::  MATRIX_1:def 3
len it = m & ( for p being FinSequence of D st p in rng it holds len p = n );
end;

definition
let D be set ;
mode Matrix of D is tabular FinSequence of D*;
end;

definition
let D be set;
redefine mode FinSequence of D -> FinSequence-like PartFunc of NAT,D;
end;

definition
let X, Y be set;
mode PartFunc of X,Y is Function-like Relation of X,Y;
end;

definition
let X, Y be set;
redefine mode Relation of X,Y -> Subset of [:X,Y:];
end;

definition
let X be set;
mode Subset of X is Element of bool X;
end;

definition
let X be set;
mode Element of X -> set means :Def2:  ::  SUBSET_1:def 2
it in X if not X is empty otherwise it is empty;
end;
```

which adds the adjective `finite` to the set of adjectives of any term which already has the adjective `Finsequence-like`. In this way the Mizar terms can get relatively large numbers of adjectives automatically, making significant portions of formal reasoning obvious to Mizar.

Mizar *structure types* are a special kind of Mizar types that are intended to encode mathematical structures, usually consisting of a carrier set and some operations on it. Typical examples are algebraic structures like groups, rings, and vector spaces, but there are also many other structures like topological and metric spaces. These structures have a special implementation in Mizar (providing additional automations) that differs from those of other Mizar types. This has both advantages and disadvantages, see [LR07] for a discussion and an alternative implementation of structures relying on the standard type mechanisms in Mizar.

2.2 Mizar second-order constructs

Mizar axiomatics is Tarski-Grothendieck set theory with strong axiom of choice. This is an extension of ZFC that adds arbitrarily large inaccessible cardinals to the universe. This extension is used to model some parts of category theory in Mizar. However, for practically all applications it is enough to think of Mizar's axiomatics as ZFC with strong choice. Particularly, the standard *Replacement (Fraenkel) Axiom scheme*[5] ("image of any set under any definable function is again a set") is used:

```
scheme :: TARSKI:sch 1
Fraenkel { A()-> set, P[set, set] }:
 ex X st for x holds x in X iff ex y st y in A() & P[y,x]
 provided for x,y,z st P[x,y] & P[x,z] holds y = z;
```

The expression `P[set, set]` here declares a "second-order" predicate variable. Its Mizar semantics is that it can be instantiated with any Mizar formula with two free variables of the type *set*. Once Mizar has to allow such second-order mechanisms for the axiomatics, it is advantageous for human authoring to allow them also for regular theorems. For example, the *Separation (Comprehension) scheme*[6] ("any definable subclass of a set is again a set"):

```
scheme :: XBOOLE_0:sch 1
 Separation { A()-> set, P[set] } :
   ex X being set st for x being set holds
                          x in X iff x in A() & P[x];
```

can be inferred from Replacement, but is much more often used in MML (490 uses of Separation vs. 24 uses of Replacement), and probably in normal mathematics too. The Replacement scheme is commonly used in mathematics to produce *Fraenkel (Abstract) terms*, i.e., terms of the form

```
{ N - M where M,N is Integer :  N < M }
```

Neither the schemes nor the Fraenkel terms can be directly expressed in pure first-order logic.

3 MPTP: Translating Mizar for Automated Reasoning tools

The translation of Mizar and MML to pure first-order logic is described in detail in [Urb03, Urb04, Urb07b, US07]. This section provides an overview of the translation using the MPTP (Mizar problems for Theorem Proving) system. In addition to the logical extensions mentioned above in Section 2, the translation from Mizar to pure first-order logic also has to deal with a number of practical issues related to the Mizar implementation, implementations of first-order ATP systems, and the most frequent uses of the translation system.

3.1 MPTP 0.1

The first version of MPTP is described in detail in [Urb03, Urb04]. This version was used for initial exploration of the usability of ATP systems on the Mizar Mathematical Library (MML). The first important number obtained was the 41% success rate of ATP-reproving of about 30000 MML theorems from other Mizar theorems and definitions selected from the corresponding MML proofs.

No previous evidence about the feasibility and usefulness of ATP methods on a very large library like MML was available prior to the experiments done with MPTP 0.1[7], sometimes leading to overly

[5] `http://www.tptp.org/MizarTPTP/Articles/tarski.html#S1`

[6] `http://www.tptp.org/MizarTPTP/Articles/subset_1.html#S1`

[7] A lot of work on MPTP was inspired by the previous work done in the ILF project [DW97] on importing Mizar. However it seemed that the project had stopped before it could finish the export of the whole MML to ATP problems, and provide some initial overall statistics of ATP success rate on MML.

pessimistic views on such a project. Therefore the goal of this first version was to relatively quickly achieve a "mostly-correct" translated version of the whole MML that would allow assessment of the potential of ATP methods for this large library. Many shortcuts and simplifications were therefore taken in this first MPTP version, naming at least the following:

- Mizar formulas were directly exported to the DFG [HKW96] syntax used by the SPASS [WBH+02] system. SPASS seemed to perform best on MPTP problems, probably because of its handling of sort theories. SPASS also has a built-in efficient clausifier [NW01], which the other efficient provers like E [Sch02] and Vampire [RV02] did not have at that time (CNF was the main category of the CASC competition until 2006).
- One simple method of handling sorts (encoding as predicates and relativization) was chosen for the export, yielding standard (untyped) first-order formulas, from which the original type information could not be recovered and used for different encodings.
- Mizar proofs were not exported. Only the lists of MML references (theorems and definitions) used for proof of each MML theorem were remembered for re-creation of ATP problems corresponding to Mizar proofs (see Section 3.2.4 for overview of the problem creation in MPTP). The proof structure and internal lemmas[8] were forgotten.
- Such lists of MML references were *theoretically* sufficient[9] as premises for re-proving of about 80% (27449 out of 33527) of theorem proofs - i.e., the Mizar proofs use only these MML references and some implicit (background) facts like type hierarchy, arithmetical evaluations, etc. In the Mizar proofs of the remaining ca 20% (6078) of theorems, Mizar *schemes* and top-level non-theorem lemmas[10] were used. These two kinds of propositions were completely ignored, making these theorems not eligible for ATP reproving.
- The export of Mizar *structure* types was incomplete (some axioms were missing), *abstract terms* were translated incorrectly, and the background theory computed for problems could sometimes be too strong, possibly leading to MML-invalid (cyclic) proofs. All these shortcuts were justified by the low frequency of such cases in MML.

Many of these simplifications made further experiments with MPTP difficult or impossible, and also made the 41% success rate uncertain (some ATP proof could succeed, and some could fail because of these simplifications). The lack of proof structure prevented measurement of ATP success rate on all internal proof lemmas, and experiments with unfolding lemmas with their own proofs. Additionally, even if only several abstract terms were translated incorrectly, during such proof unfoldings their effect could spread much wider. Experiments like finding new proofs, and cross-verification of Mizar proofs (described below) would suffer from constant doubt about the possible amount of error caused by the incorrectly translated parts of Mizar, and debugging would be very hard.

3.2 MPTP 0.2

During the work on MPTP 0.1 and other Mizar-related systems, it became clear that the old internal format used by Mizar (designed long ago, when memory and storage were expensive) was quite hard

[8] An *Internal Lemma* is a lemma proved inside a proof of a MML theorem. It can be proved either by *Simple Justification* (Mizar keyword "`by`") or it can have its own structured subproof (Mizar keywords "`proof ... end`").

[9] This means that the proof should be found by a complete ATP system with unlimited resources. As mentioned above, the *practical* ATP success was 41%.

[10] The vast majority of Mizar propositions proved at the top-level (i.e., not inside a proof of another proposition) are exported from Mizar as theorems reusable in other articles. This is however not mandatory.

to extend for new Mizar constructs and utilities. A new extensible and richer format seemed to be needed for Mizar itself, and for systems like MPTP, MMLQuery [BR03], MoMM [Urb06b], MizarMode [Urb06a, BU04], each of which had its own special-purpose exporter from Mizar doing very similar things.

This resulted in quite a large reimplementation of Mizar described in [Urb05]. Mizar started to use XML[11] natively as its internal format produced during parsing. The internal format was significantly extended, and it now contains a very complete semantically disambiguated form of a Mizar article, together with a large amount of presentational information that allows quite faithful re-creation of Mizar articles from this internal format. Because of the completeness of this format, and thanks to the widespread availability of XML parsers, the need for special-purpose Mizar exporters for various systems and the problem of their maintenance were largely eliminated. This allows quite simple combinations of Mizar-based systems, for example both the HTML and the MPTP/TPTP parts of the MizarTPTP presentation at `http://www.tptp.org/MizarTPTP` [UTSP07] are produced from the same XML form of Mizar articles, with quite simple changes to the XSL stylesheets producing the HTML.

After this necessary upgrade of Mizar, MPTP 0.2 was started from scratch, using a simple XSL stylesheet (instead of the Pascal exporter) to export the Mizar XML into an extended TPTP-like [SS98] format, and using Prolog (instead of Perl) for generating ATP problems in TPTP. In MPTP 0.1 the translated formulas were already in DFG format, and Perl (treating formulas mostly just as strings) was enough to generate the ATP problems. This is no longer true in MPTP 0.2: the extended TPTP-like format is (so far) not directly usable by ATPs, because it encodes Mizar types and abstract terms (and some other things) in a generic way, allowing different translations. Prolog was therefore needed to implement structural functions on the formulas, such as the type relativization. The MPTP 0.2 codebase has grown from ca. 900 lines of XSLTXT[12] (a compact human-friendly version of XSL) and 1500 lines of Prolog in 2005 to ca. 2000 lines of XSLTXT and 5500 lines of Prolog in 2008. The basis of the system remains the same, but the codebase has grown as a number of different functionalities (used by the projects mentioned below) have been added.

In the following subsections we summarize the main translation methods and functionalities used currently by MPTP 0.2.

3.2.1 Type translation in MPTP 0.2

MPTP extends the TPTP language to allow *parametrized types* like "`Matrix of n,m,D`", and "`tabular FinSequence of D*`" mentioned in Section 2. For example the following Mizar formula:

```
for G being infinite Graph, W1 being Walk of G
  ex W2 being Subwalk of W1 st W1=W2;
```

is translated into this extended TPTP format:

```
! [G : (~ finite & graph), W1 : walk(G)]: (? [W2 : subwalk(W1)] : (W1=W2))
```

This differs from the (proposed) typed TPTP standard which only allows atomic sorts. MPTP currently only implements the predicate encoding (relativization) of types. Type declarations with arity n are transformed into predicates with arity $n+1$, and, e.g., the above formulas becomes:

```
! [G] :
  ( ( ~ finite(G) & graph(G) )
 => ! [W1] :
```

[11] See `http://lipa.ms.mff.cuni.cz/~urban/Mizar.html` for specification of the Mizar XML format.
[12] `http://www.zanthan.com/ajm/xsltxt/`

```
    ( walk(G,W1)
   => ? [W2] :
        ( subwalk(W1,W2) & (W1=W2) ) ) )
```

The implicit type hierarchies and fixpoint automations used in Mizar for adjectives are replaced by explicit inclusion of the corresponding formulas into the ATP problems. For example, the cluster

```
cluster FinSequence-like -> finite set;
```

mentioned in Section 2 is translated as:

```
! [X] : 'FinSequence-like'(X) => finite(X)
```

and explicitly added to the ATP problem when needed (see Section 3.2.4). Because pretty Mizar symbol names are largely overloaded (there are e.g. more than 100 different meanings of the symbol '+' in MML), their unique disambiguated internal naming is used in MPTP. That means that the above formula will actually look like this:

```
! [X] : v1_finseq_1(X) => v1_finset_1(X)
```

An attempt was started by the author to provide unique descriptive names for Mizar symbols[13], and these names have already been used for creating the MPTP Challenge problems (see below). However, there are around 10000 Mizar symbols, so cooperation from other Mizar and MPTP users is needed to incrementally improve this unique descriptive naming.

3.2.2 Translation of abstract terms in MPTP 0.2

The special `all`/3 predicate is used for encoding abstract terms in MPTP 0.2. For instance the following abstract term mentioned in Section 2.2

```
{ N - M where M,N is Integer : N < M }
```

is encoded as:

```
all([M:integer,N:integer], minus(N,M), less(N,M))
```

Abstract terms are very similar to lambda terms, which are sometimes called anonymous functions. Therefore the process of removing abstract terms and inventing fresh names for them was called *deanonymization* in [Urb07b]. It seems that a similar procedure is used in the export of lambda terms in Isabelle/HOL to first-order logic, with the name *lambda-lifting*. The procedure is very similar to Skolemization, and that is why this syntactic extension could eventually become handled by standard ATP clausifiers, or even dealt with in calculi which implement delayed transformation to normal forms (e.g., tableaux or [GS03]). It means that a new functor symbol is introduced, corresponding to the abstract term in the following way:

```
![X]:(in(X,all_0_xx) <=> ?[N:integer,M:integer]:
                           (X = minus(N,M) & less(N,M))).
```

Here `all_0_xx`/0 is the newly introduced "Fraenkel" functor for the abstract term given above, the first number in it (`0`) is its arity and the second number (`xx`) just a serial numbering of such symbols with the same arity. Obviously Fraenkel functors with nonzero arity can arise if their context includes quantified variables, this is similar to Skolemization. The predicate `in`/2 (set-theoretic membership) has to be available for this encoding.

[13] `http://wiki.mizar.org/cgi-bin/twiki/view/Mizar/NiceConstructorNames`

As with Skolemization, a lot of optimizing steps can be done during deanonymization. If one abstract term is used twice, only one Fraenkel functor is necessary. This has the additional advantage that the equality of such terms is obvious, while for different Fraenkel functors the Extensionality axiom (*"two sets are equal if they contain the same elements"*) has to be used to find out that they encode the same term. Abstract terms are used often inside Mizar proofs, and proof-local constants often occur in them. A fairly efficient optimization is implemented by extracting the abstract terms from the proof context, i.e., by generalizing the proof-local constants appearing inside a term before the definition of the corresponding Fraenkel functor is created. When this is done for a whole Mizar article, the number of Fraenkel definitions can be reduced very significantly, sometimes by a factor of 10 or even 20. This extraction from the proof context turns out to be necessary for reproving Mizar theorems whose proofs contain abstract terms, because for such reproving attempts only proof-external symbols and references can be used, and Fraenkel definitions containing proof-local constants would not be accessible, possibly making the reproving task incomplete. Such article-global generation of Fraenkel definitions is a standard preprocessing step done immediately after the article is loaded (for MPTP processing) into Prolog, and the abstract terms are replaced globally in all formulas by the corresponding Fraenkel functors before any ATP problems are generated. Some of the generated Fraenkel functors are used very frequently, which suggests that they probably deserve their own proper definition as Mizar functors.

3.2.3 Translation of schemes in MPTP 0.2

The MPTP treatment of schemes (see Section 2.2) is similar to that of the abstract terms: the instances that are already present in the MML are used. This is sufficient for first-order reproving, and with a sufficiently large body of mathematics like MML (and thus sufficiently many first-order scheme instances), it could also be "reasonably sufficient" for proving new things (though obviously incomplete in general in this case).

The implementation uses Mizar (compiled with a special scheme reporting directive) to print the particular instantiations of the second-order functor and predicate variables. These second-order variables in schemes are encoded using a fresh functor or predicate symbol, so, e.g., the Separation (Comprehension) scheme (called `s1_xboole_1` in MPTP) is encoded in the extended language in this way:

```
? [B1: $true]: ![B2: $true]:
(in(B2,B1) <=> ( in(B2,f1_s1_xboole_0) & p1_s1_xboole_0(B2)))
```

i.e., in pure TPTP as:

```
? [B1]: (![B2]: (in(B2,B1) <=> ( in(B2,f1_s1_xboole_0) & p1_s1_xboole_0(B2))))
```

Here `f1_s1_xboole_0` and `p1_s1_xboole_0` are the fresh symbols encoding the second-order variables. This handling is semantically sufficient for reproving of Mizar schemes, because nothing (except their declared type restrictions) is known about these fresh first-order symbols (and therefore the proof can be instantiated to any particular first-order functors and predicates of the proper types). However, additional treatment is necessary when schemes are applied inside other proofs. In these cases, these symbols first have to be replaced with the instantiations reported by Mizar. For each scheme a number of its instances are thus obtained, again (as in the case of abstract terms) possibly containing some proof-local constants. Again, for reproving theorems, these instances have to be extracted from the proof context by generalizing the proof-local constants, and again this is done globally for a whole article and before any ATP problems are generated. Obviously, there is the objection that relying on Mizar and Prolog to carry out the second-order instantiation is a weak point of possible ATP cross-verification of Mizar proofs. The correctness of this procedure is, however, easy to check, by checking that the original scheme proofs (with the fresh first-order symbols) also work for the particular scheme instances (with the first-order symbols instantiated to the Mizar-supplied instances).

3.2.4 Problem creation and axiom selection in MPTP

There are several ways in which MPTP can create ATP problems. While the translation of the above mentioned Mizar constructs is usually fixed, the main degree of freedom is the selection of suitable premises from the translated MML for a particular conjecture. The most common MPTP task (used, e.g., for the re-proving experiments described in Section 4.1) is to generate ATP problem for a given MML theorem in such a way that all the MML theorems and definitions explicitly used in the MML proof are included, together with additional "background" formulas encoding the knowledge that can be used implicitly by Mizar. These background formulas encode the type hierarchy, definitional expansions, arithmetical evaluations, properties of Mizar functors and predicates like commutativity and antisymmetry, etc. The addition of such formulas into the ATP problems is done in a fixpoint algorithm watching the current set of symbols in the problem (initialized with the symbols contained in the formulas that are used explicitly in the MML proof), and adding these implicit facts when they might be needed. Several versions of this "enriching" algorithm exist in MPTP. The more background facts are added, the more complete the problem is (and the harder it can be for ATPs to re-prove the problem if the added axiom is redundant[14]). The default MPTP version is intended to be complete in the sense that an ATP problem corresponding to an existing Mizar proof will contain all the background axioms needed to re-play the Mizar proof by ATPs. This is useful for debugging, however stricter (heuristical) versions are available too. Additionally, this is an instance of the general "axiom selection" problem, for which specialized systems like MaLARea [Urb07a, USPV08] are being used.

4 Experiments and projects based on the MPTP

MPTP has so far been used for

- experiments with re-proving Mizar theorems and simple lemmas by ATPs from the theorems and definitions used in the corresponding Mizar proofs
- experiments with fully automated re-proving of Mizar theorems, i.e., the necessary axioms being selected fully automatically from the whole available MML
- finding new ATP proofs that are simpler than the original Mizar proofs
- ATP-based cross-verification of the Mizar proofs
- ATP-based explanation of Mizar atomic inferences
- inclusion of Mizar problems into the TPTP problem library, and unified web presentation of Mizar together with the corresponding TPTP problems
- creation of the MPTP $100 Challenges for reasoning in large theories in 2007, and subsequent creation of the MZR category of the CASC Large Theory Batch (LTB) competition in 2008
- a testbed for AI systems like MaLARea targeted at reasoning in large theories and combining inductive techniques like machine learning with deductive reasoning

[14]For example, the SPASS prover has recently re-proved the Lagrange's theorem in Mizar (`http://www.tptp.org/MizarTPTP/Articles/group_2.html#T177`) from 25 premises. However the default MPTP background-adding algorithm results in inclusion of another 135 formulas into the ATP problem, making the problem impossible to prove by existing standard ATPs.

4.1 Re-proving experiments

As mentioned in Section 3.1, the initial large-scale experiment done with MPTP 0.1 indicated that 41% of the Mizar proofs could be automatically found by ATPs, if the user provides the same theorems and definitions that are used in the Mizar proofs, plus the corresponding background formulas. As already mentioned, this number was far from certain, e.g., out of the 27449 problems tried, 625 were shown to be CounterSatisfiable by SPASS (pointing to various oversimplifications taken in MPTP 0.1). The experiment was therefore repeated with MPTP 0.2, but with only 12529 problems that come from articles that do not use internal arithmetical evaluations done by Mizar. These evaluations were not handled by MPTP 0.2 at the time these experiments were conducted, being the last (known) part of Mizar that could be blamed for possible ATP incompleteness. The E prover version 0.9 and SPASS version 2.1 were used for this experiment, with a 20s time limit (due to limited resources). The results (reported in [Urb07b]) are given in Table 2. 39% of the 12529 theorems were proved by either SPASS or E, and no countersatisfiability was found.

Table 2: Reproving of the theorems from non-numerical articles by MPTP 0.2 in 2005

description	proved	countersatisfiable	timeout or memory out	total
E 0.9	4309	0	8220	12529
SPASS 2.1	3850	0	8679	12529
together	4854	0	7675	12529

These results have thus, to a large extent, confirmed the optimistic outlook of the first measurement in MPTP 0.1. In later (so far unreported) experiments, this ATP performance has been steadily going up; see Table 3 for results from a 2007 run with a 60s timelimit. This is a result of better pruning of redundant axioms in MPTP, and also of ATP development, which obviously was influenced by the inclusion of MPTP problems into the TPTP library, forming a significant part of the FOF problems in the CASC competition since 2006. Together, in this increased timelimit, the newer versions of E and SPASS solved 6500 problems, i.e., 52% of them all. With the addition of Vampire and its customized Fampire version (which alone solves 51% of the problems), the combined success rate went up to 7694 of these problems, i.e., to 61%. The caveat is that the methods for dealing with arithmetics are becoming stronger and stronger in Mizar, and so far it is not clear how to handle them efficiently in ATPs. The MPTP problem creation for problems containing arithmetics is thus currently quite crude, and the ATP success rate on such problems will likely be significantly lower than on the nonarithmetical ones.

Table 3: Reproving of the theorems from non-numerical articles by MPTP 0.2 in 2007

description	proved	countersatisfiable	timeout or memory out	total
E 0.999	5661	0	6868	12529
SPASS 2.2	5775	0	6754	12529
E+SPASS together	6500	-	-	12529
Vampire 8.1	5110	0	7419	12529
Vampire 9	5330	0	7119	12529
Fampire 9	6411	0	6118	12529
all together	7694	-	-	12529

4.2 Finding new proofs and the AI aspects

MPTP 0.2 was also used to try to prove Mizar theorems fully automatically, i.e., the choice of premises for each theorem was done automatically, and all previously proved theorems were eligible. Because giving ATPs thousands of axioms is usually hopeless[15], the axiom selection was done by symbol-based machine learning from previously available proofs. The results (reported in [Urb07b]) are given in Table 4. 2408 of the 12529 theorems were proved either by E 0.9 or SPASS 2.1 from the axioms selected by the machine learner, and the combined success rate of this whole system was thus 19%. These experiments have not been repeated so far, as the combination of machine learning and other axiom selection methods have recently been under heavy development in the MaLARea system.

Table 4: Proving new theorems with machine learning support by MPTP 0.2 in 2005

description	proved	countersatisfiability	timeout or memory out	total
E 0.9	2167	0	10362	12529
SPASS 2.1	1543	0	10986	12529
together	2408	0	10121	12529

This experiment demonstrates a very real and quite unique benefit of large formal mathematical libraries for conducting novel integration of AI methods. As the machine learner is trained on previous proofs, it recommends relevant premises from the large library that (according to the past experience) should be useful for proving new conjectures. A variety of machine learning methods (neural nets, Bayes nets, decision trees, nearest neighbor, etc.) can be used for this, and their performance evaluated in the standard machine learning way, i.e., by looking at the actual axiom selection done by the human author in the Mizar proof, and comparing it with the selection suggested by the trained learner. However, what if the machine learner is sometimes more clever than the human, and suggests a completely different (and perhaps better) selection of premises, leading to a different proof? In such a case, the standard machine learning evaluation (i.e., comparison of the two sets of premises) will say that the two sets of premises differ too much, and thus the machine learner has failed. This is considered acceptable for machine learning, as in general, there is no deeper concept of *truth* available, there are just training and testing data. However in our domain we do have a method how to show that the trained learner was right (and possibly smarter than the human): we can run an ATP system on its axiom selection. If a proof is found, it provides a much stronger measure of correctness. Obviously, this is only true if we know that the translation from Mizar to TPTP was correct, i.e., conducting such experiments really requires that we take extra care to ensure that no oversimplifications were made in this translation.

In the above mentioned experiment, 329 of the 2408 (i.e., 14%) proofs found by ATPs used less premises than the original MML proof, often suggesting a shorter proof. An example of such proof shortening is discussed in [Urb07b], showing that the newly found proof is really valid. Instead of arguing from the first principles (definitions) like in the human proof, the combined inductive-deductive system was smart enough to find a combination of previously proved lemmas (properties) that justify the conjecture more quickly.

[15] This is changing as we go: the new CASC-LTB category will hopefully spark interest in ATP systems dealing efficiently with large numbers of unnecessary axioms.

4.3 ATP-based explanation, presentation, and cross-verification of Mizar proofs

While proofs of whole Mizar theorems can be quite hard for ATP systems, re-proving the Mizar atomic justification steps (called *Simple Justifications* in Mizar) turns out to be quite easy for ATPs. The combination of E and SPASS usually solves more than 95% of such problems, and with smarter automated methods for axiom selection a 99.8% success rate (14 unsolved problems from 6765) was achieved in [US07]. This makes it practical to use ATPs for explanation and presentation of the (not always easily understandable) Mizar simple justifications, and to construct larger systems for independent ATP-based cross-verification of (possibly very long) Mizar proofs. In [US07] such a cross-verification system is presented, using the GDV [Sut06] system (which was extended to process Jaskowski-style natural deduction proofs that make frequent use of assumptions (suppositions)). MPTP was used to translate Mizar proofs to this format, and GDV together with the E, SPASS, and MaLARea systems was used to automatically verify the structural correctness of proofs, and 99.8% of the proof steps needed for the 252 Mizar problems selected for the MPTP Challenge (see below). This provides the first practical method for independent verification of Mizar, and opens the possibility of importing Mizar proofs into other proof assistants. A web presentation allowing interaction with ATP systems and GDV verification of Mizar proofs has been set up at `http://www.tptp.org/MizarTPTP` (described in [UTSP07]), and a static presentation using the MML Query system to translate the ATP proofs back to Mizar notation exists at `http://lipa.ms.mff.cuni.cz/~urban/xmlmml/html_bytst/` (described in [UB07]). A new online service integrating these functionalities is being built at `http://octopi.mizar.org/~mptp/MizAR.html`.

4.4 Use of MPTP for ATP challenges and competitions

The first MPTP problems were included into the TPTP library in 2006, and were already used for the 2006 CASC competition [Sut07]. In 2006, the MPTP $100 Challenges[16] were created and announced. This is a set of 252 related large-theory problems needed for one half (on of two implications) of the Mizar proof of the general topological Bolzano-Weierstrass theorem. Unlike the CASC competition, the challenge had an overall timelimit (252 * 5 minutes = 21 hours) for solving the problems, allowing complementary techniques like machine learning from previous solutions to be experimented with transparently in runtime. The challenge was won a year later by the leanCoP [OB03] system, having already revealed several interesting approaches to ATP in large theories: goal-directed calculi like connection tableaux (used in leanCoP), model-based axiom selection (used, e.g., in SRASS [SP07]), and machine learning of axiom relevance (used in MaLARea). The MPTP Challenge problems were again included into the TPTP and used for the CASC competition in 2007. In 2008, the CASC-LTB (Large Theory Batch) division appeared for the first time, with a similar setting to the MPTP Challenges, and additional large-theory problems from the Cyc and SUMO ontologies. A set of 245 relatively hard Mizar problems was added to the TPTP for this purpose, coming from the most advanced parts of the Mizar library. The problems come in four versions, containing different amounts of the previously available MML theorems and definitions as axioms. The largest versions thus contain over 50000 axioms. This has practically led to the inclusion of the MML into TPTP.

4.5 Development of larger AI metasystems like MaLARea on MPTP data

In Section 4.2 it is explained how the deeply defined notion of mathematical *truth* (implemented through ATPs) can improve the evaluation of learning systems working on large semantic knowledge bases like the translated MML, and how such systems can be used to find new mathematical proofs. This is, however, only one part of the AI fun made possible by such large libraries being available to ATPs.

[16] `http://www.tptp.org/MPTPChallenge/`

Another part is that newly found proofs can be recycled, and used again for learning in such domains. This closed loop between using deductive methods to find proofs, and using inductive methods to learn from existing proofs and suggest new proof directions, is the main idea behind the MaLARea [Urb07a, USPV08] meta-system. There are many kinds of information that such an autonomous meta-system can try to use and learn. The second version of MaLARea has just started to use structural and semantic features of formulas for their characterization, and for improving the axiom selection. Extracting lemmas from proofs and adding them to the set of available premises, creating new interesting conjectures and defining new useful notions (see [Len76, Col02, Faj88] for some previous interesting work on this), finding optimal strategies for problem classes, guiding the ATPs more closely than just by selecting axioms, inventing policies for efficient governing of the overall inductive-deductive loop: all these are interesting AI tasks that become relevant in this large-theory setting, and that seem to be highly relevant for the ultimate task of *doing mathematics* and perhaps even generally *thinking* automatically.

5 Future work

There is large amount of work to be done on practically all the projects mentioned above. The MPTP translation is by no means optimal (and especially proper encoding of arithmetics needs more experiments and work). Import of ATP proofs to Mizar practically does not exist (there is a basic translator taking Otter proof objects to Mizar, however this is very distant from the readable proofs in MML). MPTP has mostly been used for offline problem generation so far, and its use for online ATP services requires further work. With sufficiently strong ATP systems, the cross-verification of the whole MML could be attempted, and work on importing detailed ATP proofs into other proof assistants could be started. The work with second-order constructs is in some sense incomplete, and either a translation to (finitely axiomatized) NBG set theory, or usage of higher-order ATPs would be interesting from this point of view. More challenges and interesting presentation tools can be developed, for example an ATP-enhanced wiki for Mizar would be quite interesting. The heuristical and machine learning methods, and combined AI metasystems, have a very long way to go, some future directions are mentioned above. This is no longer only about mathematics: all kinds of more or less formal large knowledge bases are becoming available in other sciences, and automated reasoning could become one of the strongest methods for general reasoning in sciences when sufficient amount of formal knowledge exists. Strong ATP methods for formal mathematics could also provide useful semantic filtering for larger systems for automatic formalization of mathematical papers. This is a field that has been so far deemed to be rather science fiction than a real possibility.

References

[BR03] Grzegorz Bancerek and Piotr Rudnicki. Information retrieval in MML. In *MKM*, volume 2594 of *Lecture Notes in Computer Science*, pages 119–132. Springer, 2003.

[BU04] Grzegorz Bancerek and Josef Urban. Integrated semantic browsing of the Mizar Mathematical Library for authoring Mizar articles. In *MKM*, pages 44–57, 2004.

[CLR06] Thierry Coquand, Henri Lombardi, and Marie-Françoise Roy, editors. *Mathematics, Algorithms, Proofs, 9.-14. January 2005*, volume 05021 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[CMSM04] S. Colton, A. Meier, V. Sorge, and R. McCasland. Automatic Generation of Classification Theorems for Finite Algebras. In M. Rusinowitch and D. Basin, editors, *Proceedings of the 2nd International Joint Conference on Automated Reasoning*, number 3097 in Lecture Notes in Artificial Intelligence, pages 400–414, 2004.

[Col02] S. Colton. The HR Program for Theorem Generation. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, number 2392 in Lecture Notes in Artificial Intelligence, pages 285–289. Springer-Verlag, 2002.

[DFGS99] J. Denzinger, M. Fuchs, C. Goller, and S. Schulz. Learning from Previous Proof Experience. Technical Report AR99-4, Institut für Informatik, Technische Universität München, 1999. (also to be published as a SEKI report).

[DW97] Ingo Dahn and Christoph Wernhard. First order proof problems extracted from an article in the MIZAR Mathematical Library. In Maria Paola Bonacina and Ulrich Furbach, editors, *Int. Workshop on First-Order Theorem Proving (FTP'97)*, RISC-Linz Report Series No. 97-50, pages 58–62. Johannes Kepler Universität, Linz (Austria), 1997.

[Faj88] Siemion Fajtlowicz. On conjectures of graffiti. *Discrete Mathematics*, 72(1-3):113–118, 1988.

[GS03] H. Ganzinger and J. Stuber. Superposition with Equivalence Reasoning. In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction*, number 2741 in Lecture Notes in Artificial Intelligence, pages 335–349. Springer-Verlag, 2003.

[HKW96] R. Hähnle, M. Kerber, and C. Weidenbach. Common Syntax of the DFG-Schwerpunktprogramm Deduction. Technical Report TR 10/96, Fakultät für Informatik, Universät Karlsruhe, Karlsruhe, Germany, 1996.

[HSA06] John Harrison, Konrad Slind, and Rob Arthan. Hol. In Freek Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*, pages 11–19. Springer, 2006.

[Jan91] Katarzyna Jankowska. Matrices. Abelian group of matrices. *Formalized Mathematics*, 2(4):475–480, 1991.

[Len76] D. Lenat. *An Artificial Intelligence Approach to Discovery in Mathematics*. PhD thesis, Stanford University, Stanford, USA, 1976.

[LR07] Gilbert Lee and Piotr Rudnicki. Alternative aggregates in Mizar. In Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger, editors, *Calculemus/MKM*, volume 4573 of *Lecture Notes in Computer Science*, pages 327–341. Springer, 2007.

[MP06] J. Meng and L. Paulson. Lightweight Relevance Filtering for Machine-Generated Resolution Problems. In G. Sutcliffe, R. Schmidt, and S. Schulz, editors, *Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning, 3rd International Joint Conference on Automated Reasoning*, volume 192 of *CEUR Workshop Proceedings*, pages 53–69, 2006.

[MP08] Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008.

[NP01] I. Niles and A. Pease. Towards A Standard Upper Ontology. In C. Welty and B. Smith, editors, *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems*, pages 2–9, 2001.

[NW01] A. Nonnengart and C. Weidenbach. Computing Small Clause Normal Forms. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 335–367. Elsevier Science, 2001.

[OB03] J. Otten and W. Bibel. leanCoP: Lean Connection-Based Theorem Proving. *Journal of Symbolic Computation*, 36(1-2):139–161, 2003.

[PS07] A. Pease and G. Sutcliffe. First Order Reasoning on a Large Ontology. In J. Urban, G. Sutcliffe, and S. Schulz, editors, *Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories*, 2007.

[RPG05] D. Ramachandran, Reagan P., and K. Goolsbey. First-orderized ResearchCyc: Expressiveness and Efficiency in a Common Sense Knowledge Base. In Shvaiko P., editor, *Proceedings of the Workshop on Contexts and Ontologies: Theory, Practice and Applications*, 2005.

[RT99] Piotr Rudnicki and Andrzej Trybulec. On equivalents of well-foundedness. *J. Autom. Reasoning*, 23(3-4):197–234, 1999.

[RT03] Piotr Rudnicki and Andrzej Trybulec. On the integrity of a repository of formalized mathematics. In *MKM*, volume 2594 of *Lecture Notes in Computer Science*, pages 162–174. Springer, 2003.

[Rud87] P. Rudnicki. Obvious Inferences. *Journal of Automated Reasoning*, 3(4):383–393, 1987.

[Rud92] P. Rudnicki. An Overview of the Mizar Project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 311–332, 1992.

[RV02] A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*,

15(2-3):91–110, 2002.

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

[SP07] 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.

[SS98] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[Sut06] G. Sutcliffe. Semantic Derivation Verification. *International Journal on Artificial Intelligence Tools*, 15(6):1053–1070, 2006.

[Sut07] G. Sutcliffe. The 3rd IJCAR Automated Theorem Proving Competition. *AI Communications*, 20(2):117–126, 2007.

[UB07] Josef Urban and Grzegorz Bancerek. Presenting and explaining Mizar. *Electr. Notes Theor. Comput. Sci.*, 174(2):63–74, 2007.

[Urb03] J. Urban. Translating Mizar for First Order Theorem Provers. In A. Asperti, B. Buchberger, and J.H. Davenport, editors, *Proceedings of the 2nd International Conference on Mathematical Knowledge Management*, number 2594 in Lecture Notes in Computer Science, pages 203–215. Springer-Verlag, 2003.

[Urb04] J. Urban. MPTP - Motivation, Implementation, First Experiments. *Journal of Automated Reasoning*, 33(3-4):319–339, 2004.

[Urb05] J. Urban. XML-izing Mizar: Making Semantic Processing and Presentaion of MML Easy. In M. Kohlhase, editor, *Proceedings of the 4th Integrated Conference on Mathematical Knowledge Management*, volume 3863 of *Lecture Notes in Computer Science*, pages 346–360, 2005.

[Urb06a] J. Urban. MizarMode - An Integrated Proof Assistance Tool for the Mizar Way of Formalizing Mathematics. *Journal of Applied Logic*, 4(4):414–427, 2006.

[Urb06b] Josef Urban. MoMM - fast interreduction and retrieval in large libraries of formalized mathematics. *International Journal on Artificial Intelligence Tools*, 15(1):109–130, 2006.

[Urb07a] J. Urban. MaLARea: a Metasystem for Automated Reasoning in Large Theories. In J. Urban, G. Sutcliffe, and S. Schulz, editors, *Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories*, pages 45–58, 2007.

[Urb07b] J. Urban. MPTP 0.2: Design, Implementation, and Initial Experiments. *Journal of Automated Reasoning*, 37(1-2):21–43, 2007.

[US07] J. Urban and G. Sutcliffe. ATP Cross-verification of the Mizar MPTP Challenge Problems. In N. Dershowitz and A. Voronkov, editors, *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 4790 in Lecture Notes in Artificial Intelligence, pages 546–560, 2007.

[USPV08] Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jirí Vyskocil. Malarea SG1 - machine learner for automated reasoning with semantic guidance. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 441–456, 2008.

[UTSP07] J. Urban, S. Trac, G. Sutcliffe, and Y. Puzis. Combining Mizar and TPTP Semantic Presentation Tools. In *Proceedings of the Mathematical User-Interfaces Workshop 2007*, 2007.

[VLP07] K. Verchinine, A. Lyaletski, and A. Paskevick. System for Automated Deduction (SAD): A Tool for Proof Verification. In F. Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction*, number 4603 in Lecture Notes in Artificial Intelligence, pages 398–403. Springer-Verlag, 2007.

[WBH+02] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic. SPASS Version 2.0. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, number 2392 in Lecture Notes in Artificial Intelligence, pages 275–279. Springer-Verlag, 2002.

A TLA$^+$ Proof System

Kaustuv Chaudhuri, INRIA
Damien Doligez, INRIA
Leslie Lamport, Microsoft Research
Stephan Merz, INRIA & Loria

Abstract

We describe an extension to the TLA$^+$ specification language with constructs for writing proofs and a proof environment, called the Proof Manager (PM), to checks those proofs. The language and the PM support the incremental development and checking of hierarchically structured proofs. The PM translates a proof into a set of independent proof obligations and calls upon a collection of back-end provers to verify them. Different provers can be used to verify different obligations. The currently supported back-ends are the tableau prover Zenon and Isabelle/TLA$^+$, an axiomatisation of TLA$^+$ in Isabelle/Pure. The proof obligations for a complete TLA^{+2} proof can also be used to certify the theorem in Isabelle/TLA$^+$.

1 Introduction

TLA$^+$ is a language for specifying the behavior of concurrent and distributed systems and asserting properties of those systems [11]. However, it provides no way to write proofs of those properties. We have designed an extended version of the language that allows writing proofs, and we have begun implementing a system centered around a *Proof Manager* (PM) that invokes existing automated and interactive proof systems to check those proofs. For now, the new version of TLA$^+$ is called TLA^{+2} to distinguish it from the current one. We describe here the TLA^{+2} proof constructs and the current state of the proof system.

The primary goal of TLA^{+2} and the proof system is the mechanical verification of systems specifications. The proof system must not only support the modal and temporal aspects of TLA needed to reason about system properties, but must also support ordinary mathematical reasoning in the underlying logic. Proofs in TLA^{+2} are natural deduction proofs written in a hierarchical style that we have found to be good for ordinary mathematics [9] and crucial for managing the complexity of correctness proofs of systems [6].

The PM computes proof obligations that establish the correctness of the proof and sends them to one or more back-end provers to be verified. Currently, the back-end provers are Isabelle/TLA$^+$, a faithful axiomatization of TLA$^+$ in Isabelle/Pure, and Zenon [2], a tableau prover for classical first-order logic with equality. The PM first sends a proof obligation to Zenon. If Zenon succeeds, it produces an Isar script that the PM sends to Isabelle to check. Otherwise, the PM outputs an Isar script that uses one of Isabelle's automated tactics. In both cases, the obligations are certified by Isabelle/TLA$^+$. The system architecture easily accommodates other back-end provers; if these are proof-producing, then we can use their proofs to certify the obligations in Isabelle/TLA$^+$, resulting in high confidence in the overall correctness of the proof.

The TLA^{+2} proof constructs are described in Section 2. Section 3 describes the proof obligations generated by the PM, and Section 4 describes how the PM uses Zenon and Isabelle to verify them. The conclusion summarizes what we have done and not yet done and briefly discusses related work.

Rudnicki P, Sutcliffe G., Konev B., Schmidt R., Schulz S. (eds.);
Proceedings of the Combined KEAPPA - IWIL Workshops, pp. 17-37

2 TLA$^+$ and its Proof Language

2.1 TLA

The TLA$^+$ language is based on the Temporal Logic of Actions (TLA) [10], a linear-time temporal logic. The rigid variables of TLA are called *constants* and the flexible variables are called simply *variables*. TLA assumes an underlying ordinary (non-modal) logic for constructing expressions. Operators of that logic are called *constant* operators. A *state function* is an expression built from constant operators and TLA constants and variables. The elementary (non-temporal) formulas of TLA are *actions*, which are formulas built with constant operators, constants, variables, and expressions of the form f', where f is a state function. (TLA also has an ENABLED operator that is used in expressing fairness, but we ignore it for brevity.) An action is interpreted as a predicate on pairs of states that describes a set of possible state transitions, where state functions refer to the starting state and primed state functions refer to the ending state. Because priming distributes over constant operators and because c' is equal to c for any constant c, an action can be reduced to a formula built from constant operators, constants, variables, and primed variables.

TLA is practical for describing systems because all the complexity of a specification is in the action formulas. Temporal operators are essentially used only to assert liveness properties, including fairness of system actions. Most of the work in a TLA proof is in proving action formulas; temporal reasoning occurs only in proving liveness properties and is limited to propositional temporal logic and to applying a handful of proof rules whose main premises are action formulas. Because temporal reasoning is such a small part of TLA proofs, we have deferred its implementation. The PM now handles only action formulas. We have enough experience mechanizing TLA's temporal reasoning [4] to be fairly confident that it will not be hard to extend the PM to support it.

A formula built from constant operators, constants, variables, and primed variables is valid iff it is a valid formula of the underlying logic when constants, variables, and primed variables are treated as distinct variables of the logic—that is, if v and v' are considered to be two distinct variables of the underlying logic, for any TLA variable v. Since any action formula is reducible to such a formula, action reasoning is immediately reducible to reasoning in the underlying logic. We therefore ignore variables and priming here and consider only constant formulas.

2.2 TLA$^+$

The TLA$^+$ language adds the following to the TLA logic:

- An underlying logic that is essentially ZFC set theory plus classical untyped first-order logic with Hilbert's ε [13]. The major difference between this underlying logic and traditional ZFC is that functions are defined axiomatically rather than being represented as sets of ordered pairs.
- A mechanism for defining operators, where a user-defined operator is essentially a macro that is expanded syntactically. (TLA$^+$ permits recursive function definitions, but they are translated to ordinary definitions using Hilbert's ε.)
- Modules, where one module can import definitions and theorems from other modules. A module is parameterized by its declared variables and constants, and it may be instantiated in another module by substituting expressions for its parameters. The combination of substitution and the ENABLED operator introduces some complications, but space limitations prevent us from discussing them, so we largely ignore modules in this paper.

TLA$^+$ has been extensively documented [11]. Since we are concerned only with reasoning about its underlying logic, which is a very familiar one, we do not bother to describe TLA$^+$ in any detail. All of its nonstandard notation that appears in our examples is explained.

2.3 The Proof Language

The major new feature of TLA^{+2} is its proof language. (For reasons having nothing to do with proofs, TLA^{+2} also introduces recursive operator definitions, which we ignore here for brevity.) We describe the basic proof language, omitting a few constructs that concern aspects such as module instantiation that we are not discussing. TLA^{+2} also adds constructs for naming subexpressions of a definition or theorem, which is important in practice for writing proofs but is orthogonal to the concerns of this paper.

The goal of the language is to make proofs easy to read and write for someone with no knowledge of how the proofs are being checked. This leads to a mostly declarative language, built around the uses and proofs of assertions rather than around the application of proof-search tactics. It is therefore more akin to Isabelle/Isar [17] than to more operational interactive languages such as Coq's Vernacular [16]. Nevertheless, the proof language does include a few operational constructs that can eliminate the repetition of common idioms, albeit with some loss of perspicuity.

At any point in a TLA$^+$ proof, there is a current obligation that is to be proved. The obligation contains a *context* of known facts, definitions, and declarations, and a *goal*. The obligation claims that the goal is logically entailed by the context. Some of the facts and definitions in the context are marked (explicitly or implicitly) as *usable* for reasoning, while the remaining facts and definitions are *hidden*.

Proofs are structured hierarchically. The leaf (lowest-level) proof OBVIOUS asserts that the current goal follows easily from the usable facts and definitions. The leaf proof

$$\text{BY } e_1, \ldots, e_m \text{ DEFS } o_1, \ldots, o_n$$

asserts that the current goal follows easily from the usable facts and definitions together with (i) the facts e_i that must themselves follow easily from the context and (ii) the known definitions of o_j. Whether a goal follows easily from definitions and facts depends on who is trying to prove it. For each leaf proof, the PM sends the corresponding *leaf obligation* to the back-end provers, so in practice "follows easily" means that a back-end prover can prove it. A non-leaf proof is a sequence of *steps*, each consisting of a begin-step token and a proof construct. For some constructs (including a simple assertion of a proposition) the step takes a subproof, which may be omitted. The final step in the sequence simply asserts the current goal, which is represented by the token QED. A begin-step token is either a *level token* of the form $\langle n \rangle$ or a *label* of the form $\langle n \rangle l$, where n is a level number that is the same for all steps of this non-leaf proof, and l is an arbitrary name. The hierarchical structure is deduced from the level numbers of the begin-step tokens, a higher level number beginning a subproof.

Some steps make declarations or definitions or change the current goal and do not require a proof. Other steps make assertions that become the current goals for their proofs. An omitted proof (or one consisting of the token OMITTED) is considered to be a leaf proof that instructs the assertion to be accepted as true. Of course, the proof is then incomplete. From a logical point of view, an omitted step is the same as an additional assumption added to the theorem; from a practical point of view, it doesn't have to be lifted from its context and stated at the start. Omitted steps are intended to be used only in the intermediate stages of writing a proof.

Following a step that makes an assertion (and the step's proof), until the end of the current proof (after the QED step), the contexts contain that assertion in their sets of known facts. The assertion is marked usable iff the begin-step token is a level token; otherwise it can be referred to by its label in a BY proof or made usable with a USE step.

The hierarchical structure of proofs not only aids in reading the finished proof but is also quite useful in incrementally writing proofs. The steps of a non-leaf proof are first written with all proofs but that of the QED step omitted. After checking the proof of the QED step, the proofs omitted for other steps in this or earlier levels are written in any order. When writing the proof, one may discover facts that are needed in the proofs of multiple steps. Such a fact is then added to the proof as an earlier step, or

added at a higher level. It can also be removed from the proof of the theorem and proved separately as a lemma. However, the hierarchical proof language encourages facts relevant only for a particular proof to be kept within the proof, making the proof's structure easier to see and simplifying maintenance of the proof. For correctness proofs of systems, the first few levels of the hierarchy are generally determined by the structure of the formula to be proved—for example, the proof that a formula implies a conjunction usually consists of steps asserting that it implies each conjunct.

As an example, we incrementally construct a hierarchical proof of Cantor's theorem, which states that there is no surjective function from a set to its powerset. It is written in TLA$^+$ as:

THEOREM $\forall S : \forall f \in [S \to \text{SUBSET } S] : \exists A \in \text{SUBSET } S : \forall x \in S : f[x] \neq A$

where function application is written using square brackets, SUBSET S is the powerset of S, and $[S \to T]$ is the set of functions from S to T.

The statement of the theorem is the current goal for its top-level proof. A goal of the form $\forall v : e$ is proved by introducing a generic constant and proving the formula obtained by substituting it for the bound identifier. We express this as follows, using the ASSUME/PROVE construct of TLA^{+2}:

```
THEOREM ∀S : ∀f ∈ [S → SUBSET S] : ∃A ∈ SUBSET S : ∀x ∈ S : f[x] ≠ A
  ⟨1⟩1. ASSUME NEW S,
                NEW f ∈ [S → SUBSET S]
        PROVE ∃A ∈ SUBSET S : ∀x ∈ S : f[x] ≠ A
  ⟨1⟩2. QED BY ⟨1⟩1
```

Although we could have used labels such as ⟨1⟩*one* and ⟨1⟩*last* instead of ⟨1⟩1 and ⟨1⟩2, we have found that proofs are easier to read when steps at the same level are labeled with consecutive numbers. One typically starts using consecutive step numbers and then uses labels like ⟨3⟩2a for inserting additional steps. When the proof is finished, steps are renumbered consecutively. (A planned user interface will automate this renumbering.)

Step ⟨1⟩1 asserts that for any constants S and f with $f \in [S \to \text{SUBSET } S]$, the proposition to the right of the PROVE is true. More precisely, the current context for the (as yet unwritten) proof of ⟨1⟩1 contains the declarations of S and f and the usable fact $f \in [S \to \text{SUBSET } S]$, and the PROVE assertion is its goal. The QED step states that the original goal (the theorem) follows from the assertion in step ⟨1⟩1.

We tell the PM to check this (incomplete) proof, which it does by having the back-end provers verify the proof obligation for the QED step. The verification succeeds, and we now continue by writing the proof of ⟨1⟩1. (Had the verification failed because ⟨1⟩1 did not imply the current goal, we would have caught the error before attempting to prove ⟨1⟩1, which we expect to be harder to do.)

We optimistically start with the proof OBVIOUS, but it is too hard for the back-end to prove, and the PM reports a timeout. Often this means that a necessary fact or definition in the context is hidden and we merely have to make it usable with a USE step or a BY proof. In this case we have no such hidden assumptions, so we must refine the goal into simpler goals with a non-leaf proof. We let this proof have level 2 (we can use any level greater than 1). Since the goal itself is existentially quantified, we must supply a witness. In this case, the witness is the classic diagonal set, which we call T.

```
  ⟨1⟩1. ASSUME NEW S,
                NEW f ∈ [S → SUBSET S]
        PROVE ∃A ∈ SUBSET S : ∀x ∈ S : f[x] ≠ A
    ⟨2⟩1. DEFINE T ≜ {z ∈ S : z ∉ f[z]}
    ⟨2⟩2. ∀x ∈ S : f[x] ≠ T
    ⟨2⟩3. QED BY ⟨2⟩2
```

Because definitions made within a proof are usable by default, the definition of T is usable in the proofs of ⟨2⟩2 and ⟨2⟩3. Once again, the proof of the QED step is automatically verified, so all that remains is to prove ⟨2⟩2. (The DEFINE step requires no proof.)

The system accepts OBVIOUS as the proof of $\langle 2\rangle 2$ because the only difficulty in the proof of $\langle 1\rangle 1$ is finding the witness. However, suppose we want to add another level of proof for the benefit of a human reader. The universal quantification is proved as above, by introducing a fresh constant:

$\langle 2\rangle 2.\ \forall x \in S : f[x] \neq T$
  $\langle 3\rangle 1.$ ASSUME NEW $x \in S$ PROVE $f[x] \neq T$
  $\langle 3\rangle 2.$ QED BY $\langle 3\rangle 1$

Naturally, the QED step is verified. Although the system accepts OBVIOUS as the proof of $\langle 3\rangle 1$ (remember that it could verify $\langle 2\rangle 2$ by itself), we can provide more detail with yet another level of proof. We write this proof the way it would seem natural to a person—by breaking it into two cases:

$\langle 3\rangle 1.$ ASSUME NEW $x \in S$ PROVE $f[x] \neq T$
  $\langle 4\rangle 1.$ CASE $x \in T$
  $\langle 4\rangle 2.$ CASE $x \notin T$
  $\langle 4\rangle 3.$ QED BY $\langle 4\rangle 1, \langle 4\rangle 2$

The (omitted) proof of the CASE statement $\langle 4\rangle 1$ has as its goal $f[x] \neq T$ and has the additional usable fact $x \in T$ in its context.

We continue refining the proof in this way, stopping with an OBVIOUS or BY proof when a goal is obvious enough for the back-end prover or for a human reader, depending on who the proof is being written for. A BY statement can guide the prover or the human reader by listing helpful obvious consequences of known facts. For example, the proof of $\langle 4\rangle 1$ might be BY $x \notin f[x]$. The proof is now finished: it contains no omitted sub-proofs. For reference, the complete text of the proof is given in Appendix B.

Our experience writing hand proofs makes us expect that proofs of systems could be ten or more levels deep, with the first several levels dictated by the structure of the property to be proved. Our method of numbering steps makes such proofs manageable, and we are not aware of any good alternative.

This example illustrates how the proof language supports the hierarchical, non-linear, and incremental development of proofs. The proof writer can work on the most problematic unproved steps first, leaving the easier ones for later. Finding that a step cannot be proved (for example, because it is invalid) may require changing other steps, making proofs of those other steps wasted effort. We intend to provide an interface to the PM that will make it easy for the user to indicate which proofs should be checked and will avoid unnecessarily rechecking proofs.

The example also shows how already-proved facts are generally not made usable, but are invoked explicitly in BY proofs. Global definitions are also hidden by default and the user must explicitly make them usable. This makes proofs easier to read by telling the reader what facts and definitions are being used to prove each step. It also helps constrain the search space for an automated back-end prover, leading to more efficient verification. Facts and definitions can be switched between usable and hidden by USE and HIDE steps, which have the same syntax as BY. As noted above, omitting the label from a step's starting token (for example, writing $\langle 4\rangle$ instead of $\langle 4\rangle 2$) makes the fact it asserts usable. This might be done for compactness at the lowest levels of a proof.

The example also indicates how the current proof obligation at every step of the proof is clear, having been written explicitly in a parent assertion. This clear structure comes at the cost of introducing many levels of proof, which can be inconvenient. One way of avoiding these extra levels is by using an assertion of the form SUFFICES A, which asserts that proving A proves the current goal, and makes A the new current goal in subsequent steps. In our example proof, one level in the proof of step $\langle 2\rangle 2$ can be eliminated by writing the proof as:

$\langle 2\rangle 2.\ \forall x \in S : f[x] \neq T$
  $\langle 3\rangle 1.$ SUFFICES ASSUME NEW $x \in S$ PROVE $f[x] \neq T$

PROOF OBVIOUS
⟨3⟩2. CASE $x \in T$
⟨3⟩3. CASE $x \notin T$
⟨3⟩4. QED BY ⟨3⟩2, ⟨3⟩3

where the proofs of the CASE steps are the same as before. The SUFFICES statement changes the current goal of the level-3 proof to $f[x] \neq T$ after adding a declaration of x and the usable fact $x \in S$ to the context. This way of proving a universally quantified formula is sufficiently common that TLA^{+2} provides a TAKE construct that allows the SUFFICES assertion ⟨3⟩1 and its OBVIOUS proof to be written TAKE $x \in S$.

There is a similar construct, WITNESS $f \in S$ for proving an existentially quantified goal $\exists x \in S : e$, which changes the goal to $e[x := f]$. For implicational goals $e \Rightarrow f$, the construct HAVE e changes the goal to f. No other constructs in the TLA^{+2} proof language change the form of the current goal. We advise that these constructs be used only at the lowest levels of the proof, since the new goal they create must be derived instead of being available textually in a parent assertion. (As a check and an aid to the reader, one can at any point insert a redundant SUFFICES step that simply asserts the current goal.)

The final TLA^{+2} proof construct is PICK $x : e$, which introduces a new symbol x that satisfies e. The goal of the proof of this PICK step is $\exists x : e$, and it changes the context of subsequent steps by adding a declaration of x and the fact e. A more formal summary of the language appears in Appendix A.

The semantics of a TLA^{+2} proof is independent of any back-end prover. Different provers will have different notions of what "follows easily", so an OBVIOUS proof may be verified by one prover and not another. In practice, many provers such as Isabelle must be directed to use decision procedures or special tactics to prove some assertions. For this purpose, special standard modules will contain dummy theorems for giving directives to the PM. Using such a theorem (with a USE step or BY proof) will cause the PM not to use it as a fact, but instead to generate special directives for back-end provers. It could even cause the PM to use a different back-end prover. (If possible, the dummy theorem will assert a true fact that suggests the purpose of the directive.) For instance, using the theorem *Arithmetic* might be interpreted as an instruction to use a decision procedure for integers. We hope that almost all uses of this feature will leave the TLA^{+2} proof independent of the back-end provers. The proof will not have to be changed if the PM is reconfigured to replace one decision procedure with a different one.

3 Proof Obligations

The PM generates a separate *proof obligation* for each leaf proof and orchestrates the back-end provers to verify these obligations. Each obligation is independent and can be proved individually. If the system cannot verify an obligation within a reasonable amount of time, the PM reports a failure. The user must then determine if it failed because it depends on hidden facts or definitions, or if the goal is too complex and needs to be refined with another level of proof. (Hiding facts or definitions might also help to constrain the search space of the back-end provers.)

When the back-end provers fail to find a proof, the user will know which obligation failed—that is, she will be told the obligation's usable context and goal and the leaf proof from which it was generated. We do not yet know if this will be sufficient in practice or if the PM will need to provide the user with more information about why an obligation failed. For example, many SAT and SMT solvers produce counterexamples for an unprovable formula that can provide useful debugging information.

The PM will also mediate the *certification* of the TLA^{+2} theorem in a formal axiomatization of TLA^{+2} in a trusted logical framework, which in the current design is Isabelle/TLA$^+$ (described in Section 4.2). Although the PM is designed generically and can support other similar frameworks, for the rest of this paper we will limit our attention to Isabelle/TLA$^+$. Assuming that Isabelle/TLA$^+$ is sound, once it has

certified a theorem we know that an error is possible only if the PM incorrectly translated the statement of the theorem into Isabelle/TLA$^+$.

After certifying the proof obligations generated for the leaf proofs, called the *leaf obligations*, certification of the theorem itself is achieved in two steps. First, the PM generates a *structure lemma* (and its Isabelle/TLA$^+$ proof) that states simply that the collection of leaf obligations implies the theorem. Then, the PM generates a proof of the theorem using the already-certified obligations and structure lemma. If Isabelle accepts that proof, we are assured that the translated version of the theorem is true in Isabelle/TLA$^+$, regardless of any errors made by the PM.

Of course, we expect the PM to be correct. We now explain why it should be by describing how it generates the leaf obligations from the proof of a theorem. (Remember that we are considering only TLA^{+2} formulas with no temporal operators.) Formally, a theorem in TLA^{+2} represents a closed proof obligation in the TLA^{+2} meta-logic of the form $(\Gamma \Vdash e)$, where Γ is a *context* containing all the declarations, definitions, facts (previous assumptions or theorems) and the assumptions introduced in the theorem using an ASSUME clause (if present), and e is a TLA^{+2} formula that is the *goal* of the theorem.

A closed obligation $(\Gamma \Vdash e)$ is *true* if e is entailed by Γ in the formal semantics of TLA$^+$ [11]. It is said to be *provable* if we have a proof of e from Γ in Isabelle/TLA$^+$. Because we assume Isabelle/TLA$^+$ to be sound, we consider any provable obligation to be true. A *claim* is a sentence of the form $\pi : (\Gamma \Vdash e)$, where π is a TLA^{+2} proof. This claim represents the verification task that π is a proof of the proof obligation $(\Gamma \Vdash e)$. The PM generates the leaf obligations of a claim by recursively traversing its proof, using its structure to refine the obligation of the claim. For a non-leaf proof, each proof step modifies the context or the goal of its obligation to produce an obligation for its following step, and the final QED step proves the final form of the obligation. More precisely, every step defines a *transformation*, written $\sigma.\tau : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f)$, which states that the *input* obligation $(\Gamma \Vdash e)$ is *refined* to the obligation $(\Delta \Vdash f)$ by the step $\sigma.\tau$. A step is said to be *meaningful* if the input obligation matches the form of the step. (An example of a meaningless claim is one that involves a TAKE step whose input obligation does not have a universally quantified goal.) A claim is meaningful if every step in it is meaningful.

The recursive generation of leaf obligations for meaningful claims and transformations is specified using inference rules, with the interpretation that the leaf obligations generated for the claim or transformation at the conclusion of a rule is the union of those generated by the claims and transformations in the premises of the rule. For example, the following rule is applied to generate the leaf obligations for a claim $\pi : (\Gamma \Vdash e)$ when π is a sequence of n steps, for $n > 1$.

$$\frac{\sigma_1.\tau_1 : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f) \qquad \sigma_2.\tau_2 \;\; \cdots \;\; \sigma_n.\tau_n : (\Delta \Vdash f)}{\sigma_1.\tau_1 \;\; \sigma_2.\tau_2 \;\; \cdots \;\; \sigma_n.\tau_n : (\Gamma \Vdash e)}$$

The leaf obligations of the claim in the conclusion are the union of those of the claim and transformation in the premises. As an example of leaf obligations generated by a transformation, here is a rule for the step $\sigma.\tau$ where σ is the begin-step level token $\langle n \rangle$ and τ is the proposition p with proof π.

$$\frac{\pi : (\Gamma, [\neg e] \Vdash p)}{\langle n \rangle .\ p \text{ PROOF } \pi : (\Gamma \Vdash e) \longrightarrow (\Gamma, p \Vdash e)}$$

The rule concludes that the refinement in this step is to add p to the context of the obligation, assuming that the sub-proof π is able to establish it. The leaf obligations generated by this transformation are the same as those of the claim in the premise of the rule. The goal e is negated and added to the context as a hidden fact (the square brackets indicate hiding). We can use $\neg e$ in a BY proof or USE statement, and doing so can simplify subproofs. (Because we are using classical logic, it is sound to add $\neg e$ to the known facts in this way.) The full set of such rules for every construct in the TLA^{+2} proof language is given in appendix A.

A claim is said to be *complete* it its proof contains no omitted subproofs. Starting from a complete meaningful claim, the PM first generates its leaf obligations and *filters* the hidden assumptions from their contexts. (Filtration amounts to deleting hidden facts and replacing hidden operator definitions with declarations.) The PM then asks the back-end provers to find proofs of the filtered obligations, which are used to certify the obligations in Isabelle/TLA$^+$. The PM next writes an Isar proof of the obligation of the complete meaningful claim that uses its certified filtered leaf obligations. The following meta-theorem (proved in Appendix A.4) ensures that the PM can do this for all complete meaningful claims.

Theorem 1 (Structural Soundness Theorem)**.** *If* $\pi : (\Gamma \Vdash e)$ *is a complete meaningful claim and every leaf obligation it generates is provable after filtering hidden assumptions , then* $(\Gamma \Vdash e)$ *is provable.*

Isabelle/TLA$^+$ then uses this proof to certify the obligation of the claim. From the assumptions that the Isabelle/TLA$^+$ axiomatization is faithful to the semantics of TLA^{+2} and that the embedding of TLA^{+2} into Isabelle/TLA$^+$ is sound, it follows that the obligation is true.

4 Verifying Proof Obligations

Once the PM generates the leaf obligations, it must send them to the back-end provers. The one non-obvious part of doing this is deciding whether definitions should be expanded by the PM or by the prover. This is discussed in Section 4.1. We then describe the state of our two current back-end provers, Isabelle/TLA$^+$ and Zenon.

4.1 Expanding Definitions

Expansion of usable definitions cannot be left entirely to the back-end prover. The PM itself must do it for two reasons:

- It must check that the current goal has the right form for a TAKE, WITNESS, or HAVE step to be meaningful, and this can require expanding definitions.
- The encoding of TLA$^+$ in the back-end prover's logic would be unsound if a modal operator like prime ($'$) were encoded as a non-modal operator. Hence, encoding a definition like $O(x) \triangleq x'$ as an ordinary definition in the prover's logic would be unsound. All instances of such operators must be removed by expanding their definitions before a leaf obligation is sent to the back-end prover. Such operator definitions seldom occur in actual TLA$^+$ specifications, but the PM must be able to deal with them.

Another reason for the PM to handle definition expansion is that the Isabelle/TLA$^+$ object logic does not provide a direct encoding of definitions made within proofs. We plan to reduce the amount of trusted code in the PM by lambda-lifting all usable definitions out of each leaf obligation and introducing explicit operator definitions using Isabelle's meta equality ($\equiv$). These definitions will be expanded before interacting with Isabelle.

4.2 Isabelle/TLA$^+$

The core of TLA^{+2} is being encoded as a new object logic Isabelle/TLA$^+$ in the proof assistant Isabelle [14]. One of Isabelle's distinctive features that similar proof assistants such as Coq [16] or HOL [7, 8] lack is genericity with respect to different logics. The base system Isabelle/Pure provides the trusted kernel and a framework in which the syntax and proof rules of object logics can be defined. We have chosen to encode TLA^{+2} as a separate object logic rather than add it on top of one of the existing logics (such as ZF or HOL). This simplifies the translation and makes it easier to interpret the

error messages when Isabelle fails to prove obligations. A strongly typed logic such as HOL would have been unsuitable for representing TLA^{+2}, which is untyped. Isabelle/ZF might seem like a natural choice, but differences between the way it and TLA$^+$ define functions and tuples would have made the encoding awkward and would have prevented us from reusing existing theories. Fortunately, the genericity of Isabelle helped us not only to define the new logic, but also to instantiate the main automated proof methods, including rewriting, resolution- and tableau provers, and case-based and inductive reasoning. Adding support for more specialized reasoning tools such as proof-producing SAT solvers [5] or SMT solvers such as haRVey [3] will be similarly helped by existing generic interfaces.

The current encoding supports only a core subset of TLA^{+2}, including propositional and first-order logic, elementary set theory, functions, and the construction of natural numbers. Support for arithmetic, strings, tuples, sequences, and records is now being added; support for the modal part of TLA^{+2} (variables, priming, and temporal logic) will be added later. Nevertheless, the existing fragment can already be used to test the interaction of the PM with Isabelle and other back-end provers. As explained above, Isabelle/TLA$^+$ is used both as a back-end prover and to check proof scripts produced by other back-end provers such as Zenon. If it turns out to be necessary, we will enable the user to invoke one of Isabelle's automated proof methods (such as `auto` or `blast`) by using a dummy theorem, as explained at the end of Section 2.3. If the method succeeds, one again obtains an Isabelle theorem. Of course, Isabelle/TLA$^+$ can also be used independently of the PM, which is helpful when debugging tactics.

4.3 Zenon

Zenon [2] is a tableau prover for classical first-order logic with equality that was initially designed to output formal proofs checkable by Coq [16]. Zenon outputs proofs in an automatically-checkable format and it is easily extensible with new inference rules. One of its design goals is predictability in solving simple problems, rather than high performance in solving some hard problems. These characteristics make it well-suited to our needs.

We have extended Zenon to output Isar proof scripts for Isabelle/TLA$^+$ theorems, and the PM uses Zenon as a back-end prover, shipping the proofs it produces to Isabelle to certify the obligation. We have also extended Zenon with direct support for the TLA^{+2} logic, including definitions and rules about sets and functions. Adding support in the form of rules (instead of axioms) is necessary because some rules are not expressible as first-order axioms, notably the rules about the set constructs:

$$\frac{e \in S \qquad P[x := e]}{e \in \{x \in S : P\}}\ \mathit{subsetOf} \qquad \frac{\exists y \in S : e = d[x := y]}{e \in \{d : x \in S\}}\ \mathit{setOfAll}$$

Even for the rules that are expressible as first-order axioms, adding them as rules makes the proof search procedure much more efficient in practice. The most important example is extensionality: when set extensionality and function extensionality are added as axioms, they apply to every equality deduced by the system, and pollute the search space with large numbers of irrelevant formulas. By adding them as rules instead, we can use heuristics to apply them only in cases where they have some chance of being useful.

Adding support for arithmetic, strings, tuples, sequences, and records will be done in parallel with the corresponding work on Isabelle/TLA$^+$, to ensure that Zenon will produce proof scripts that Isabelle/TLA$^+$ will be able to check. Temporal logic will be added later. We also plan to interface Zenon with Isabelle, so it can be called by a special Isabelle tactic the same way other tools are. This will simplify the PM by giving it a uniform interface to the back-end provers. It will also allow using Zenon as an Isabelle tactic independently of TLA$^+$.

5 Conclusions and Future Work

We have presented a hierarchically structured proof language for TLA$^+$. It has several important features that help in managing the complexity of proofs. The hierarchical structure means that changes made at any level of a proof are contained inside that level, which helps construct and maintain proofs. Leaf proofs can be omitted and the resulting incomplete proof can be checked. This allows different parts of the proof to be written separately, in a non-linear fashion. The more traditional linear proof style, in which steps that have not yet been proved can be used only if explicitly added as hypotheses, encourages proofs that use many separate lemmas. Such proofs lack the coherent structure of a single hierarchical proof.

The proof language lets the user freely and repeatedly make facts and definitions usable or hidden. Explicitly stating what is being used to prove each step makes the proof easier for a human to understand. It also aids a back-end prover by limiting its search for a proof to ones that use only necessary facts.

There are other declarative proof languages that are similar to TLA^{+2}. Isar [17] is one such language, but it has significant differences that encourage a different style of proof development. For example, it provides an *accumulator* facility to avoid explicit references to proof steps. This is fine for short proofs, but in our experience does not work well for long proofs that are typical of algorithm verification that TLA^{+2} targets. Moreover, because Isabelle is designed for interactive use, the effects of the Isar proof commands are not always easily predictable, and this encourages a linear rather than hierarchical proof development style. The Focal Proof Language [1] is essentially a subset of the TLA^{+2} proof language. Our experience with hierarchical proofs in Focal provides additional confidence in the attractiveness of our approach. We know of no declarative proof language that has as flexible a method of using and hiding facts and definitions as that of TLA^{+2}.

The PM transforms a proof into a collection of proof obligations to be verified by a back-end prover. Its current version handles proofs of theorems in the non-temporal fragment of TLA$^+$ that do not involve module instantiation (importing of modules with substitution). Even with this limitation, the system can be useful for many engineering applications. We are therefore concentrating on making the PM and its back-end provers handle this fragment of TLA$^+$ effectively before extending them to the complete language. The major work that remains to be done on this is to complete the Zenon and Isabelle inference rules for reasoning about the built-in constant operators of TLA$^+$. There are also a few non-temporal aspects of the TLA^{+2} language that the PM does not yet handle, such as subexpression naming. We also expect to extend the PM to support additional back-end provers, including decision procedures for arithmetic and for propositional temporal logic.

We do not anticipate that any major changes will be needed to the TLA^{+2} proof language. We do expect some minor tuning as we get more experience using it. For example, we are not sure whether local definitions should be usable by default. A graphical user interface is being planned for the TLA$^+$ tools, including the PM. It will support the non-linear development of proofs that the language and the proof system allow.

References

[1] P. Ayrault, M. Carlier, D. Delahaye, C. Dubois, D. Doligez, L. Habib, T. Hardin, M. Jaume, C. Morisset, F. Pessaux, R. Rioboo, and P. Weis. Secure software within Focal. In *Computer & Electronics Security Applications Rendez-vous*, December 2008.

[2] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In N. Dershowitz and A. Voronkov, editors, *Proc. 14th LPAR*, pages 151–165, 2007.

[3] David Déharbe, Pascal Fontaine, Silvio Ranise, and Christophe Ringeissen. Decision procedures for the formal analysis of software. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *Intl. Coll. Theoretical Aspects of Computing (ICTAC 2007)*, volume 4281 of *Lecture Notes in Computer Science*, pages 366–370, Tunis, Tunisia, 2007. Springer. See also `http://harvey.loria.fr/`.

[4] Urban Engberg, Peter Grønning, and Leslie Lamport. Mechanical verification of concurrent systems with TLA. In G. v. Bochmann and D. K. Probst, editors, *Proc. 4th CAV*, volume 663 of *Lecture Notes in Computer Science*, pages 44–55. Springer-Verlag, June 1992.

[5] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In Holger Hermanns and Jens Palsberg, editors, *Proc. 12th TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181, Vienna, Austria, 2006. Springer Verlag.

[6] Eli Gafni and Leslie Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, 2003.

[7] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL: a theorem proving environment for higher-order logic*. Cambridge University Press, 1993.

[8] John Harrison. The HOL Light theorem prover. `http://www.cl.cam.ac.uk/~jrh13/hol-light/index.html`.

[9] Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, August 1993.

[10] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[11] Leslie Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2003.

[12] Leslie Lamport. TLA^{+2}: A preliminary guide. Draft manuscript, April 2008. `http://research.microsoft.com/users/lamport/tla/tla2-guide.pdf`.

[13] A. C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*. Gordon and Breach, New York, 1969.

[14] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, Berlin, Germany, 1994.

[15] Piotr Rudnicki. An overview of the mizar project. In *Workshop on Types for Proofs and Programs*, Gothenburg, Sweden, 1992. Bastad. `http://www.mizar.org`.

[16] The Coq Development Team (Project TypiCal). The Coq proof assistant reference manual, 2008. `http://coq.inria.fr/V8.1pl3/refman/index.html`.

[17] Makarius Wenzel. The Isabelle/Isar reference manual, June 2008. `http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/isar-ref.pdf`.

A Details of the PM

We shall now give a somewhat more formal specification of the PM and prove the key Structural Soundness Theorem 1. We begin with a quick summary of the abstract syntax of TLA^{+2} proofs, ignoring the stylistic aspects of their concrete representation. (See [12] for a more detailed presentation of the proof language.)

Definition 2 (TLA^{+2} Proof Language)**.** *TLA^{+2}* proofs, non-leaf proofs, proof steps *and* begin-step *tokens have the following syntax, where n ranges over natural numbers, l over labels, e over expressions, Φ over lists of expressions, o over operator definitions, Ψ over sets of operator names, $\vec{\beta}$ over lists of binders (i.e., constructs of the form x and x ∈ e used to build quantified expressions), and α over expressions or* ASSUME . . . PROVE *forms.*

(Proofs)	π	::=	OBVIOUS | OMITTED | BY Φ DEFS Ψ | Π
(Non-leaf proofs)	Π	::=	σ. QED PROOF π
			| σ. τ Π
(Proof steps)	τ	::=	USE Φ DEFS Ψ | HIDE Φ DEFS Ψ | DEFINE o

$$\begin{array}{lll} & & \mid \textsc{have}\ e \mid \textsc{take}\ \vec{\beta} \mid \textsc{witness}\ \Phi \\ & & \mid \alpha\ \textsc{proof}\ \pi \mid \textsc{suffices}\ \alpha\ \textsc{proof}\ \pi \mid \textsc{pick}\ \vec{\beta} : e\ \textsc{proof}\ \pi \\ \textit{(Begin-step tokens)} & \sigma & ::= \langle n \rangle \mid \langle n \rangle\, l \end{array}$$

A proof that is not a non-leaf proof is called a leaf proof. *The level numbers of a non-leaf proof must all be the same, and those in the subproof of a step (that is, the π in α PROOF π, etc.) must be strictly greater than that of the step itself.*

A.1 The Meta-Language

The PM uses proofs in the TLA^{+2} proof language (Definition 2) to manipulate constructs in the meta-language of TLA^{+2}. This meta-language naturally has no representation in TLA^{+2} itself; we define its syntax formally as follows.

Definition 3 (Meta-Language)**.** *The TLA^{+2} meta-language consists of* obligations*,* assumptions *and* definables *with the following syntax, where e ranges over TLA^{+2} expressions, x and o over TLA^{+2} identifiers, and $\vec{x}$ over lists of TLA^{+2} identifiers.*

$$\begin{array}{llll} \textit{(Obligations)} & \phi & ::= (h_1, \ldots, h_n \Vdash e) & (n \geq 0) \\ \textit{(Assumptions)} & h & ::= \textsc{new}\, x \mid o \triangleq \delta \mid \phi \mid [o \triangleq \delta] \mid [\phi] & \\ \textit{(Definables)} & \delta & ::= \phi \mid \textsc{lambda}\ \vec{x} : e & \end{array}$$

The expression after $\Vdash$ in an obligation is called its goal*. An assumption written inside square brackets [ ] is said to be* hidden*; otherwise it is* usable*. For any assumption h, we write $\overline{h}$ (read: h* made usable*) to stand for h with its brackets removed if it is a hidden assumption, and to stand for h if it is not hidden. A list of assumptions is called a* context*, with the empty context written as $\bullet$; we let Γ, Δ and Ω range over contexts, with Γ, Δ standing for the context that is the concatenation of Γ and Δ. The context $\overline{\Gamma}$ is Γ with all its hidden assumptions made usable. The obligation $(\bullet \Vdash e)$ is written simply as e. The assumptions* NEW *x*, $o \triangleq \delta$ *and* $[o \triangleq \delta]$ bind *the identifiers x and o respectively. We write $x \in \Gamma$ if x is bound in Γ and $x \notin \Gamma$ if x is not bound in Γ. The context Γ, h is considered syntactically well-formed iff h does not bind an identifier already bound in Γ.*

An obligation is a statement that its goal follows from the assumptions in its context. TLA^{+2} already defines such a statement using ASSUME ... PROVE, but the contexts in such statements have no hidden assumptions or definitions. (To simplify the presentation, we give the semantics of a slightly enhanced proof language where proof steps are allowed to mention obligations instead of just TLA^{+2} ASSUME ... PROVE statements.) We define an embedding of obligations into Isabelle/TLA$^+$ propositions, which we take as the ultimate primitives of the TLA^{+2} meta-logic.

Definition 4. *The Isabelle/TLA$^+$* embedding $(-)_{\text{Isa}}$ *of obligations, contexts and definables is as follows:*

$$\begin{array}{rcl} (\Gamma \Vdash e)_{\text{Isa}} & = & (\overline{\Gamma})_{\text{Isa}}\ e \\ (\textsc{lambda}\ \vec{x} : e)_{\text{Isa}} & = & \lambda \vec{x}.\ e \end{array} \qquad \begin{array}{rcl} (\bullet)_{\text{Isa}} & = & \\ (\Gamma, \textsc{new}\, x)_{\text{Isa}} & = & (\Gamma)_{\text{Isa}} \bigwedge x. \\ (\Gamma, o \triangleq \delta)_{\text{Isa}} & = & (\Gamma)_{\text{Isa}} \bigwedge o. (o \equiv (\delta)_{\text{Isa}}) \Longrightarrow \\ (\Gamma, \phi)_{\text{Isa}} & = & (\Gamma)_{\text{Isa}}\ ((\phi)_{\text{Isa}}) \Longrightarrow \end{array}$$

For example, $(\textsc{new}\, P, [(\textsc{new}\, x \Vdash P(x))] \Vdash \forall x : P(x))_{\text{Isa}} = \bigwedge P.\ (\bigwedge x.\ P(x)) \Longrightarrow \forall x : P(x)$. Note that usable and hidden assumptions are treated identically for the provability of an obligation.

The embedding of ordinary TLA^{+2} expressions is the identity because Isabelle/TLA$^+$ contains TLA^{+2} expressions as part of its object syntax. Thus, we do not have to trust the embedding of ordinary TLA^{+2}

expressions, just that of the obligation language. In practice, some aspects of TLA^{+2} expressions, such as the indentation-sensitive conjunction and disjunction lists, are sent by the PM to Isabelle using an indentation-insensitive encoding.While Isabelle/TLA$^+$ can implicitly generalize over the free identifiers in a lemma, we shall be explicit about binding and consider obligations provable only if they are closed.

Definition 5 (Well-Formed Obligations)**.** *The obligation* $(\Gamma \Vdash e)$ *is said to be* well-formed *iff it is closed and* $(\Gamma \Vdash e)_{\text{Isa}}$ *is a well-typed proposition of Isabelle/TLA*$^+$*.*

Definition 6 (Provability)**.** *The obligation* $(\Gamma \Vdash e)$ *is said to be* provable *iff it is well-formed and* $(\Gamma \Vdash e)_{\text{Isa}}$ *is certified by the Isabelle kernel to follow from the axioms of the Isabelle/TLA*$^+$ *object logic.*

We trust Isabelle/TLA$^+$ to be sound with respect to the semantics of TLA^{+2}, and therefore provability to imply truth. Formally, we work under the following *trust* axiom.

Axiom 7 (Trust)**.** *If* ϕ *is provable, then it is true.*

We state a number of useful facts about obligations (which are all theorems in Isabelle/TLA$^+$), omitting their trivial proofs. The last one (Fact 13) is true because TLA$^+$ is based on classical logic.

Fact 8 (Definition)**.** *If* $(\Gamma, \text{NEW}\, o, \Delta \Vdash e)$ *is provable, then* $(\Gamma, o \triangleq \delta, \Delta \Vdash e)$ *is provable if it is well-formed.*

Fact 9 (Weakening)**.** *If* $(\Gamma, \Delta \Vdash e)$ *is provable, then* $(\Gamma, h, \Delta \Vdash e)$ *is provable if it is well-formed.*

Fact 10 (Expansion)**.** *If* $(\Gamma, o \triangleq \delta, \Delta \Vdash e)$ *is provable, then* $(\Gamma, o \triangleq \delta, \Delta[o := \delta] \Vdash e[o := \delta])$ *is provable.*

Fact 11 (Strengthening)**.** *If* $(\Gamma, \text{NEW}\, o, \Delta \Vdash e)$ *or* $(\Gamma, o \triangleq \delta, \Delta \Vdash e)$ *is provable and* o *is not free in* $(\Delta \Vdash e)$*, then* $(\Gamma, \Delta \Vdash e)$ *is provable.*

Fact 12 (Cut)**.** *If* $(\Gamma, \Delta \Vdash e)$ *is provable and* $(\Gamma, (\Delta \Vdash e), \Omega \Vdash f)$ *is provable, then* $(\Gamma, \Omega \Vdash f)$ *is provable.*

Fact 13. *If* $(\Gamma, \neg e, \Delta \Vdash e)$ *is provable, then* $(\Gamma, \Delta \Vdash e)$ *is provable.*

The USE/HIDE DEFS steps change the visibility of definitions in a context (Definition 14 below). Note that changing the visibility of a definition does not affect the provability of an obligation because the Isabelle embedding (Definition 4) makes all hidden definitions usable.

Definition 14. *If* Γ *is a context and* Ψ *a set of operator names, then:*

1. Γ with Ψ made usable*, written* $\Gamma \,\text{USING}\, \Psi$*, is constructed from* Γ *by replacing all assumptions of the form* $[o \triangleq \delta]$ *in* Γ *with* $o \triangleq \delta$ *for every* $o \in \Psi$*.*
2. Γ with Ψ made hidden*, written* $\Gamma \,\text{HIDING}\, \Psi$*, is constructed from* Γ *by replacing all assumptions of the form* $o \triangleq \delta$ *in* Γ *with* $[o \triangleq \delta]$ *for every* $o \in \Psi$*.*

A sequence of binders $\vec{\beta}$ in the TLA^{+2} expressions $\forall \vec{\beta} : e$ or $\exists \vec{\beta} : e$ can be reflected as assumptions.

Definition 15 (Binding Reflection)**.** *If* $\vec{\beta}$ *is a list of binders with each element of the form* x *or* $x \in e$*, then the* reflection *of* $\vec{\beta}$ *as assumptions, written* $\left\|\vec{\beta}\right\|$*, is given inductively as follows.*

$$\left\|\cdot\right\| = \cdot \qquad \left\|\vec{\beta}, x\right\| = \left\|\vec{\beta}\right\|, \text{NEW}\, x \qquad \left\|\vec{\beta}, x \in e\right\| = \left\|\vec{\beta}\right\|, \text{NEW}\, x, x \in e$$

A.2 Interpreting Proofs

Let us recall some definitions from section 3.

Definition 16 (Claims and Transformations)**.** *A* claim *is a judgement of the form* $\pi : (\Gamma \Vdash e)$ *where* π *is a TLA*$^{+2}$ *proof. A* transformation *is a judgement of the form* $\sigma.\tau : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f)$ *where* σ *is a begin-step token and* τ *a proof step. A claim (respectively, transformation) is said to be* complete *if its proof (respectively, proof step) does not contain any occurrence of the leaf proof* OMITTED*.*

The PM generates leaf obligations for a claim using two mutually recursive procedures, *checking* and *transformation*, specified below using the formalism of a *primitive derivation*.

Definition 17. *A* primitive derivation *is a derivation constructed using inferences of the form*

$$\frac{\mathcal{D}_1 \qquad \cdots \qquad \mathcal{D}_n}{E} \qquad (n \geq 0)$$

where E is either a claim or a transformation, and $\mathcal{D}_1, \ldots, \mathcal{D}_n$ *are primitive derivations or obligations. An obligation at the leaf of a primitive derivation is called a* leaf obligation*.*

Definition 18 (Checking and Transformation)**.** *The primitive derivations of a claim or transformation are constructed using the following* checking *and* transformation *rules.*

1. Checking *rules*

$$\frac{(\Gamma \Vdash e)}{\text{OBVIOUS} : (\Gamma \Vdash e)}\ \text{OBVIOUS} \qquad \frac{}{\text{OMITTED} : (\Gamma \Vdash e)}\ \text{OMITTED}$$

$$\frac{\langle 0 \rangle .\ \text{USE}\ \Phi\ \text{DEFS}\ \Psi : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f) \qquad (\Delta \Vdash f)}{\text{BY}\ \Phi\ \text{DEFS}\ \Psi : (\Gamma \Vdash e)}\ \text{BY}$$

$$\frac{\pi : (\Gamma \Vdash e)}{\sigma.\ \text{QED PROOF}\ \pi : (\Gamma \Vdash e)}\ \text{QED} \qquad \frac{\sigma.\tau : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f) \qquad \Pi : (\Delta \Vdash f)}{\sigma.\tau\ \ \Pi : (\Gamma \Vdash e)}\ \text{non-QED}$$

2. Transformation

$$\frac{\sigma.\ \text{USE}\ \Phi : (\Gamma\ \text{USING}\ \Psi \Vdash e) \longrightarrow (\Delta \Vdash f)}{\sigma.\ \text{USE}\ \Phi\ \text{DEFS}\ \Psi : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f)}\ \text{USE DEFS}$$

$$\frac{\sigma.\ \text{HIDE}\ \Phi : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f)}{\sigma.\ \text{HIDE}\ \Phi\ \text{DEFS}\ \Psi : (\Gamma \Vdash e) \longrightarrow (\Delta\ \text{HIDING}\ \Psi \Vdash f)}\ \text{HIDE DEFS}$$

$$\frac{}{\sigma.\ \text{DEFINE}\ o \triangleq \delta : (\Gamma \Vdash e) \longrightarrow (\Gamma, [o \triangleq \delta] \Vdash e)}\ \text{DEFINE}\ (o \notin \Gamma)$$

$$\frac{}{\sigma.\ \text{USE}\bullet : (\Gamma \Vdash e) \longrightarrow (\Gamma \Vdash e)}\ \text{USE}_0 \qquad \frac{}{\sigma.\ \text{HIDE}\bullet : (\Gamma \Vdash e) \longrightarrow (\Gamma \Vdash e)}\ \text{HIDE}_0$$

$$\frac{\sigma.\ \text{USE}\ \Phi : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f) \qquad (\overline{\Delta}, \Gamma_0 \Vdash e_0)}{\sigma.\ \text{USE}\ \Phi, (\Gamma_0 \Vdash e_0) : (\Gamma \Vdash e) \longrightarrow (\Delta, (\Gamma_0 \Vdash e_0) \Vdash f)}\ \text{USE}_1$$

$$\frac{\sigma.\ \text{HIDE}\ \Phi : (\Gamma_0, [\phi], \Gamma_1 \Vdash e) \longrightarrow (\Delta \Vdash f)}{\sigma.\ \text{HIDE}\ \Phi, \phi : (\Gamma_0, \phi, \Gamma_1 \Vdash e) \longrightarrow (\Delta \Vdash f)}\ \text{HIDE}_1$$

$$\frac{}{\sigma.\ \text{TAKE}\bullet : (\Gamma \Vdash e) \longrightarrow (\Gamma \Vdash e)}\ \text{TAKE}_0 \qquad \frac{}{\sigma.\ \text{WITNESS}\bullet : (\Gamma \Vdash e) \longrightarrow (\Gamma \Vdash e)}\ \text{WITNESS}_0$$

$$\frac{\sigma.\ \text{TAKE}\ \vec{\beta} : (\Gamma, \text{NEW}\ u \Vdash e[x := u]) \longrightarrow (\Delta \Vdash f)}{\sigma.\ \text{TAKE}\ u, \vec{\beta} : (\Gamma \Vdash \forall x : e) \longrightarrow (\Delta \Vdash f)}\ \text{TAKE}_1$$

$$\frac{(\Gamma \Vdash S \subseteq T) \qquad \sigma.\ \textsc{take}\ \vec{\beta} : (\Gamma, \textsc{new}\, u, u \in T \Vdash e[x := u]) \longrightarrow (\Delta \Vdash f)}{\sigma.\ \textsc{take}\ u \in T, \vec{\beta} : (\Gamma \Vdash \forall x \in S : e) \longrightarrow (\Delta \Vdash f)}\ \textsc{take}_2$$

$$\frac{\sigma.\ \textsc{witness}\ \Omega : (\Gamma \Vdash e[x := w]) \longrightarrow (\Delta \Vdash f)}{\sigma.\ \textsc{witness}\ w, \Omega : (\Gamma \Vdash \exists x : e) \longrightarrow (\Delta \Vdash f)}\ \textsc{witness}_1$$

$$\frac{(\Gamma \Vdash T \subseteq S) \qquad (\Gamma \Vdash w \in T) \qquad \sigma.\ \textsc{witness}\ \Omega : (\Gamma, w \in T \Vdash e[x := w]) \longrightarrow (\Delta \Vdash f)}{\sigma.\ \textsc{witness}\ w \in T, \Omega : (\Gamma \Vdash \exists x \in S : e) \longrightarrow (\Delta \Vdash f)}\ \textsc{witness}_2$$

$$\frac{(\Gamma, e \Vdash g)}{\sigma.\ \textsc{have}\ g : (\Gamma \Vdash e \Rightarrow f) \longrightarrow (\Gamma, g \Vdash f)}\ \textsc{have}$$

$$\frac{\pi : (\Gamma, [\neg e], \Delta \Vdash f)}{\langle n \rangle .\ (\Delta \Vdash f)\ \textsc{proof}\ \pi : (\Gamma \Vdash e) \longrightarrow (\Gamma, (\Delta \Vdash f) \Vdash e)}\ \textsc{assert}_1$$

$$\frac{\pi : (\Gamma, \langle n \rangle l \triangleq (\Delta \Vdash f), [\neg e], \Delta \Vdash f)}{\langle n \rangle l.\ (\Delta \Vdash f)\ \textsc{proof}\ \pi : (\Gamma \Vdash e) \longrightarrow (\Gamma, \langle n \rangle l \triangleq (\Delta \Vdash f), [\langle n \rangle l] \Vdash e)}\ \textsc{assert}_2$$

$$\frac{\sigma.\ (g \Vdash e)\ \textsc{proof}\ \pi : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f)}{\sigma.\ \textsc{case}\ g\ \textsc{proof}\ \pi : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f)}\ \textsc{case}$$

$$\frac{\pi : (\Gamma, (\Delta \Vdash f) \Vdash e)}{\langle n \rangle .\ \textsc{suffices}\ (\Delta \Vdash f)\ \textsc{proof}\ \pi : (\Gamma \Vdash e) \longrightarrow (\Gamma, [\neg e], \Delta \Vdash f)}\ \textsc{suffices}_1$$

$$\frac{\pi : (\Gamma, \langle n \rangle l \triangleq (\Delta \Vdash f), [\langle n \rangle l] \Vdash e)}{\langle n \rangle l.\ \textsc{suffices}\ (\Delta \Vdash f)\ \textsc{proof}\ \pi : (\Gamma \Vdash e) \longrightarrow (\Gamma, \langle n \rangle l \triangleq (\Delta \Vdash f), [\neg e], \Delta \Vdash f)}\ \textsc{suffices}_2$$

$$\frac{\pi : \left(\Gamma \Vdash \exists \vec{\beta} : p\right)}{\sigma.\ \textsc{pick}\ \vec{\beta} : p\ \textsc{proof}\ \pi : (\Gamma \Vdash e) \longrightarrow \left(\Gamma, \left\|\vec{\beta}\right\|, p \Vdash e\right)}\ \textsc{pick}$$

The inference rules in the above definition are deterministic: the conclusion of each rule uniquely determines the premises. However, the rules are partial; for example, there is no rule that concludes a transformation of the form $\sigma.\textsc{take}\ x \in S : (\Gamma \Vdash B \wedge C) \longrightarrow (\Delta \Vdash f)$.

Definition 19. *A claim or a transformation is said to be* meaningful *if it has a primitive derivation.*

Definition 20 (Generating Leaf Obligations)**.** *A meaningful claim or transformation is said to* generate *the leaf obligations of its primitive derivation.*

In the rest of this appendix we limit our attention to complete meaningful claims and transformations.

A.3 Correctness

If the leaf obligations generated by a complete meaningful claim are provable, then the obligation in the claim itself ought to be provable. In this section we prove this theorem by analysis of the checking and transformation rules.

Definition 21 (Provability of Claims and Transformation)**.**

1. *The claim $\pi : (\Gamma \Vdash e)$ is* provable *iff it is complete and meaningful and the leaf obligations it generates are all provable.*
2. *The transformation $\sigma.\tau : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f)$ is* provable *iff it is complete and meaningful and the leaf obligations it generates are all provable.*

Theorem 22 (Correctness)**.**

(1) If $\pi : (\Gamma \Vdash e)$ is provable, then $(\Gamma \Vdash e)$ is provable.

(2) If $\sigma.\tau : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f)$ *is provable and* $(\Delta \Vdash f)$ *is provable, then* $(\Gamma \Vdash e)$ *is provable.*

Proof. Let $\mathcal{D}$ be the primitive derivation for the claim in (1) and let $\mathcal{E}$ be the primitive derivation for the transformation in (2). The proof will be by lexicographic induction on the structures of $\mathcal{D}$ and $\mathcal{E}$, with a provable transformation allowed to justify a provable claim.

⟨1⟩1. If $\pi : (\Gamma \Vdash e)$ is provable, then $(\Gamma \Vdash e)$ is provable.

⟨2⟩1. *Case* π is OBVIOUS, *i.e.*, $\mathcal{D} = \dfrac{(\Gamma \Vdash e)}{\text{OBVIOUS} : (\Gamma \Vdash e)}$ OBVIOUS. *Obvious*

⟨2⟩2. *Case* π is OMITTED is impossible because $\pi : (\Gamma \Vdash e)$ is complete.

⟨2⟩3. *Case* π is BY Φ DEFS Ψ, *i.e.*,

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{E}_0\\ \langle 0\rangle\,.\ \text{USE}\ \Phi\ \text{DEFS}\ \Psi : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f) \end{array} \qquad (\Delta \Vdash f)}{\text{BY}\ \Phi\ \text{DEFS}\ \Psi : (\Gamma \Vdash e)}\ \text{BY}.$$

⟨3⟩1. $(\Delta \Vdash f)$ is provable *By* Definition 21.
⟨3⟩2. *Qed* *By* ⟨3⟩1, i.h. (inductive hypothesis) for $\mathcal{E}_0$.

⟨2⟩4. *Case* π is σ. QED PROOF π_0, *i.e.*, $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_0\\ \pi_0 : (\Gamma \Vdash e)\end{array}}{\sigma.\ \text{QED PROOF}\ \pi_0 : (\Gamma \Vdash e)}$ QED. *By* i.h. for $\mathcal{D}_0$.

⟨2⟩5. *Case* π is $\sigma.\tau\ \ \Pi$, *i.e.*,

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{E}_0\\ \sigma.\tau : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f)\end{array} \qquad \begin{array}{c}\mathcal{D}_0\\ \Pi : (\Delta \Vdash f)\end{array}}{\sigma.\tau\ \ \Pi : (\Gamma \Vdash e)}\ \text{non-QED}.$$

⟨3⟩1. $(\Delta \Vdash f)$ is provable *By* i.h. for $\mathcal{D}_0$.
⟨3⟩3. *Qed* *By* ⟨3⟩1, i.h. for $\mathcal{E}_0$.

⟨2⟩6. *Qed* *By* ⟨2⟩1, . . . , ⟨2⟩5.

⟨1⟩2. If $\sigma.\tau : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f)$ is provable and $(\Delta \Vdash f)$ is provable, then $(\Gamma \Vdash e)$ is provable.

⟨2⟩1. *Case* τ is USE Φ DEFS Ψ, *i.e.*,

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}_0\\ \sigma.\ \text{USE}\ \Phi : (\Gamma\ \text{USING}\ \Psi \Vdash e) \longrightarrow (\Delta \Vdash f)\end{array}}{\sigma.\ \text{USE}\ \Phi\ \text{DEFS}\ \Psi : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f)}\ \text{USE DEFS}.$$

⟨3⟩1. $(\Gamma\ \text{USING}\ \Psi \Vdash e)$ is provable *By* i.h. for $\mathcal{E}_0$.
⟨3⟩2. *Qed* *By* ⟨3⟩1, Definition 14.

⟨2⟩2. *Case* τ is HIDE Φ DEFS Ψ, *i.e.*,

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}_0\\ \sigma.\ \text{HIDE}\ \Phi : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f)\end{array}}{\sigma.\ \text{HIDE}\ \Phi\ \text{DEFS}\ \Psi : (\Gamma \Vdash e) \longrightarrow (\Delta\ \text{HIDING}\ \Psi \Vdash f)}\ \text{HIDE DEFS}.$$

⟨3⟩1. $(\Delta \Vdash f)$ is provable *By* provability of $(\Delta\ \text{HIDING}\ \Psi \Vdash f)$ and Definition 14.
⟨3⟩2. *Qed* *By* ⟨3⟩1, i.h. for $\mathcal{E}_0$.

⟨2⟩3. *Case* τ is DEFINE $o \triangleq \delta$ with $o \notin \Gamma$, *i.e.*,

$$\mathcal{E} = \frac{}{\sigma.\ \text{DEFINE}\ o \triangleq \delta : (\Gamma \Vdash e) \longrightarrow (\Gamma, [o \triangleq \delta] \Vdash e)}\ \text{DEFINE}.$$

⟨3⟩1. o is not free in e *By* $o \notin \Gamma$ and closedness of $(\Gamma \Vdash e)$.
⟨3⟩2. *Qed* *By* ⟨3⟩1, strengthening (Fact 11).

⟨2⟩4. *Case* τ is USE•, *i.e.*, $\mathcal{E} = \dfrac{}{\sigma.\ \text{USE}\bullet : (\Gamma \Vdash e) \longrightarrow (\Gamma \Vdash e)}$ USE$_0$. *Obvious*

⟨2⟩5. *Case* τ is HIDE•, *i.e.*, $\mathcal{E} = \dfrac{}{\sigma.\ \text{HIDE}\bullet : (\Gamma \Vdash e) \longrightarrow (\Gamma \Vdash e)}\ \text{HIDE}_0$. *Obvious*

⟨2⟩6. *Case* τ is USE Φ, ϕ, *i.e.*,

$$\mathcal{E} = \dfrac{\begin{matrix}\mathcal{E}_0\\ \sigma.\ \text{USE}\ \Phi : (\Gamma \Vdash e) \longrightarrow (\Delta_0 \Vdash f)\end{matrix} \qquad (\overline{\Delta_0}, \Gamma_0 \Vdash e_0)}{\sigma.\ \text{USE}\ \Phi, (\Gamma_0 \Vdash e_0) : (\Gamma \Vdash e) \longrightarrow (\Delta_0, (\Gamma_0 \Vdash e_0) \Vdash f)}\ \text{USE}_1$$

⟨3⟩1. $(\overline{\Delta_0}, \Gamma_0 \Vdash e_0)$ is provable *By* Definition 21.
⟨3⟩2. $(\Delta_0, \Gamma_0 \Vdash e_0)$ is provable *By* ⟨3⟩1, Definition 4.
⟨3⟩3. $(\Delta_0 \Vdash f)$ is provable *By* provability of $(\Delta_0, (\Gamma_0 \Vdash e_0) \Vdash f)$, ⟨3⟩2, cut (Fact 12).
⟨3⟩4. *Qed* *By* ⟨3⟩3, i.h. for $\mathcal{E}_0$

⟨2⟩7. *Case* τ is HIDE Φ, ϕ, *i.e.*,

$$\mathcal{E} = \dfrac{\begin{matrix}\mathcal{E}_0\\ \sigma.\ \text{HIDE}\ \Phi : (\Gamma_0, [\phi], \Gamma_1 \Vdash e) \longrightarrow (\Delta \Vdash f)\end{matrix}}{\sigma.\ \text{HIDE}\ \Phi, \phi : (\Gamma_0, \phi, \Gamma_1 \Vdash e) \longrightarrow (\Delta \Vdash f)}\ \text{HIDE}_1.$$

⟨3⟩1. $(\Gamma_0, [\phi], \Gamma_1 \Vdash e)$ is provable *By* provability of $(\Delta \Vdash f)$, i.h. for $\mathcal{E}_0$.
⟨3⟩2. *Qed* *By* ⟨3⟩1, $(\Gamma_0, [\phi], \Gamma_1 \Vdash e)_{\text{Isa}} = (\Gamma_0, \phi, \Gamma_1 \Vdash e)_{\text{Isa}}$ (Definition 4).

⟨2⟩8. *Case* τ is TAKE•, *i.e.*, $\mathcal{E} = \dfrac{}{\sigma.\ \text{TAKE}\bullet : (\Gamma \Vdash e) \longrightarrow (\Gamma \Vdash e)}\ \text{TAKE}_0$. *Obvious*

⟨2⟩9. *Case* τ is WITNESS•, *i.e.*, $\mathcal{E} = \dfrac{}{\sigma.\ \text{WITNESS}\bullet : (\Gamma \Vdash e) \longrightarrow (\Gamma \Vdash e)}\ \text{WITNESS}_0$. *Obvious*

⟨2⟩10. *Case* τ is TAKE $u, \vec{\beta}$, *i.e.*,

$$\mathcal{E} = \dfrac{\begin{matrix}\mathcal{E}_0\\ \sigma.\ \text{TAKE}\ \vec{\beta} : (\Gamma, \text{NEW}\ u \Vdash e[x := u]) \longrightarrow (\Delta \Vdash f)\end{matrix}}{\sigma.\ \text{TAKE}\ u, \vec{\beta} : (\Gamma \Vdash \forall x : e) \longrightarrow (\Delta \Vdash f)}\ \text{TAKE}_1.$$

⟨3⟩1. $(\Gamma, \text{NEW}\ u \Vdash e[x := u])$ is provable *By* i.h. for $\mathcal{E}_0$.
⟨3⟩2. *Qed* *By* ⟨3⟩1 and predicate logic.

⟨2⟩11. *Case* τ is $\sigma.$ TAKE $u \in T$, *i.e.*,

$$\mathcal{E} = \dfrac{\begin{matrix}\mathcal{E}_0\\ (\Gamma \Vdash S \subseteq T) \qquad \sigma.\ \text{TAKE}\ \vec{\beta} : (\Gamma, \text{NEW}\ u, u \in T \Vdash e[x := u]) \longrightarrow (\Delta \Vdash f)\end{matrix}}{\sigma.\ \text{TAKE}\ u \in T, \vec{\beta} : (\Gamma \Vdash \forall x \in S : e) \longrightarrow (\Delta \Vdash f)}\ \text{TAKE}_2.$$

⟨3⟩1. $(\Gamma, \text{NEW}\ u, u \in T \Vdash e[x := u])$ is provable *By* i.h on $\mathcal{E}_0$.
⟨3⟩2. $(\Gamma, \text{NEW}\ u, u \in S \Vdash u \in T)$ is provable
  ⟨4⟩1. $(\Gamma, \text{NEW}\ u \Vdash S \subseteq T)$ is provable *By* Definition 21, weakening (Fact 9).
  ⟨4⟩2. *Qed* *By* ⟨4⟩1, Definition of $\subseteq$.
⟨3⟩3. $(\Gamma, \text{NEW}\ u, u \in S \Vdash e[x := u])$ is provable *By* ⟨3⟩1, ⟨3⟩2, cut (Fact 12).
⟨3⟩4. *Qed* *By* ⟨3⟩3 and predicate logic.

⟨2⟩12. *Case* τ is WITNESS w, Ω, *i.e.*,

$$\mathcal{E} = \dfrac{\begin{matrix}\mathcal{E}_0\\ \sigma.\ \text{WITNESS}\ \Omega : (\Gamma \Vdash e[x := w]) \longrightarrow (\Delta \Vdash f)\end{matrix}}{\sigma.\ \text{WITNESS}\ w, \Omega : (\Gamma \Vdash \exists x : e) \longrightarrow (\Delta \Vdash f)}\ \text{WITNESS}_1.$$

⟨3⟩1. $(\Gamma \Vdash e[x := w])$ is provable *By* i.h. for $\mathcal{E}_0$.
⟨3⟩2. *Qed* *By* ⟨3⟩1.

⟨2⟩13. *Case* τ is WITNESS $w \in T, \Omega$ and:

$$\mathcal{E} = \dfrac{\begin{matrix}\mathcal{E}_0\\ (\Gamma \Vdash T \subseteq S) \qquad (\Gamma \Vdash w \in T) \qquad \sigma.\ \text{WITNESS}\ \Omega : (\Gamma, w \in T \Vdash e[x := w]) \longrightarrow (\Delta \Vdash f)\end{matrix}}{\sigma.\ \text{WITNESS}\ w \in T, \Omega : (\Gamma \Vdash \exists x \in S : e) \longrightarrow (\Delta \Vdash f)}\ \text{WITNESS}_2.$$

$\langle 3\rangle 1$. $(\Gamma, w \in T \Vdash e[x := w])$ is provable — *By* i.h. for $\mathcal{E}_0$.
$\langle 3\rangle 2$. $(\Gamma \Vdash w \in T)$ is provable — *By* Definition 21.
$\langle 3\rangle 3$. $(\Gamma \Vdash e[x := w])$ is provable — *By* $\langle 3\rangle 1$, $\langle 3\rangle 2$, cut (Fact 12).
$\langle 3\rangle 4$. $(\Gamma \Vdash w \in S)$ is provable
  $\langle 4\rangle 1$. $(\Gamma, w \in T \Vdash w \in S)$ is provable — *By* Definition 21, Definition of $\subseteq$.
  $\langle 4\rangle 2$. *Qed* — *By* $\langle 4\rangle 1$, $\langle 3\rangle 2$, cut (Fact 12).
$\langle 3\rangle 5$. *Qed* — *By* $\langle 3\rangle 3$, $\langle 3\rangle 4$, and predicate logic.

$\langle 2\rangle 14$. τ is HAVE g, *i.e.*,

$$\mathcal{E} = \frac{(\Gamma, e \Vdash g)}{\sigma.\ \text{HAVE}\ g : (\Gamma \Vdash e \Rightarrow f) \longrightarrow (\Gamma, g \Vdash f)}\ \text{HAVE}.$$

$\langle 3\rangle 1$. $(\Gamma, e, g \Vdash f)$ is provable — *By* weakening (Fact 9).
$\langle 3\rangle 2$. $(\Gamma, e \Vdash g)$ is provable — *By* Definition 21.
$\langle 3\rangle 3$. $(\Gamma, e \Vdash f)$ is provable — *By* $\langle 3\rangle 1$, $\langle 3\rangle 2$, cut (Fact 12).
$\langle 3\rangle 4$. $(\Gamma \Vdash e \Rightarrow f)$ is provable — *By* $\langle 3\rangle 3$.

$\langle 2\rangle 15$. $\sigma.\tau$ is $\langle n\rangle.\ (\Omega \Vdash g)$ PROOF π, *i.e.*,

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{D}_0 \\ \pi : (\Gamma, [\neg e], \Omega \Vdash g)\end{array}}{\langle n\rangle.\ (\Omega \Vdash g)\ \text{PROOF}\ \pi : (\Gamma \Vdash e) \longrightarrow (\Gamma, (\Omega \Vdash g) \Vdash e)}\ \text{ASSERT}_1.$$

$\langle 3\rangle 1$. $(\Gamma, [\neg e], (\Omega \Vdash g) \Vdash e)$ is provable — *By* weakening (Fact 9).
$\langle 3\rangle 2$. $(\Gamma, [\neg e], \Omega \Vdash g)$ is provable — *By* i.h. for $\mathcal{D}_0$.
$\langle 3\rangle 3$. $(\Gamma, [\neg e] \Vdash e)$ is provable — *By* $\langle 3\rangle 1$, $\langle 3\rangle 2$, cut (Fact 12).
$\langle 3\rangle 4$. *Qed* — *By* $\langle 3\rangle 3$, Fact 13.

$\langle 2\rangle 16$. *Case* $\sigma.\tau$ is $\langle n\rangle l.\ (\Omega \Vdash g)$ PROOF π, *i.e.*,

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{D}_0 \\ \pi : (\Gamma, \langle n\rangle l \triangleq (\Omega \Vdash g), [\neg e], \Omega \Vdash g)\end{array}}{\langle n\rangle l.\ (\Omega \Vdash g)\ \text{PROOF}\ \pi : (\Gamma \Vdash e) \longrightarrow (\Gamma, \langle n\rangle l \triangleq (\Omega \Vdash g), [\langle n\rangle l] \Vdash e)}\ \text{ASSERT}_2.$$

$\langle 3\rangle 1$. $(\Gamma, \langle n\rangle l \triangleq (\Omega \Vdash g), [\neg e], [\langle n\rangle l] \Vdash e)$ is provable
  By provability of $(\Gamma, \langle n\rangle l \triangleq (\Omega \Vdash g), [\langle n\rangle l] \Vdash e)$, weakening (Fact 9).
$\langle 3\rangle 2$. $(\Gamma, \langle n\rangle l \triangleq (\Omega \Vdash g), [\neg e], [(\Omega \Vdash g)] \Vdash e)$ is provable — *By* $\langle 3\rangle 1$, expansion (Fact 10).
$\langle 3\rangle 3$. $(\Gamma, \langle n\rangle l \triangleq (\Omega \Vdash g), [\neg e], \Omega \Vdash g)$ is provable — *By* i.h. for $\mathcal{D}_0$.
$\langle 3\rangle 4$. $(\Gamma, \langle n\rangle l \triangleq (\Omega \Vdash g), [\neg e] \Vdash e)$ is provable — *By* $\langle 3\rangle 2$, $\langle 3\rangle 3$, cut (Fact 12).
$\langle 3\rangle 5$. $(\Gamma, [\neg e] \Vdash e)$ is provable — *By* $\langle 3\rangle 4$, strengthening (Fact 11).
$\langle 3\rangle 6$. *Qed* — *By* $\langle 3\rangle 5$, Fact 13.

$\langle 2\rangle 17$. τ is CASE g PROOF π, *i.e.*,

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}_0 \\ \sigma.\ (g \Vdash e)\ \text{PROOF}\ \pi : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f)\end{array}}{\sigma.\ \text{CASE}\ g\ \text{PROOF}\ \pi : (\Gamma \Vdash e) \longrightarrow (\Delta \Vdash f)}\ \text{CASE}.$$

By i.h. for $\mathcal{E}_0$.

$\langle 2\rangle 18$. τ is $\langle n\rangle.$ SUFFICES $(\Omega \Vdash g)$ PROOF π, *i.e.*,

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{D}_0 \\ \pi : (\Gamma, (\Omega \Vdash g) \Vdash e)\end{array}}{\langle n\rangle.\ \text{SUFFICES}\ (\Delta \Vdash f)\ \text{PROOF}\ \pi : (\Gamma \Vdash e) \longrightarrow (\Gamma, [\neg e], \Omega \Vdash g)}\ \text{SUFFICES}_1.$$

$\langle 3\rangle 1$. $(\Gamma, [\neg e], (\Omega \Vdash g) \Vdash e)$ is provable — *By* i.h. for $\mathcal{D}_0$, weakening (Fact 9).
$\langle 3\rangle 2$. $(\Gamma, [\neg e] \Vdash e)$ is provable — *By* provability of $(\Gamma, [\neg e], \Omega \Vdash g)$, $\langle 3\rangle 1$, cut (Fact 12).
$\langle 3\rangle 3$. *Qed* — *By* $\langle 3\rangle 2$, Fact 13.

⟨2⟩19. $\sigma.\tau$ is $\langle n\rangle l.$ SUFFICES $(\Omega \Vdash g)$ PROOF π, *i.e.*,

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{D}_0\\ \pi : (\Gamma, \langle n\rangle l \triangleq (\Omega \Vdash g), [\langle n\rangle l] \Vdash e)\end{array}}{\langle n\rangle l.\ \text{SUFFICES}\ (\Omega \Vdash g)\ \text{PROOF}\ \pi : (\Gamma \Vdash e) \longrightarrow (\Gamma, \langle n\rangle l \triangleq (\Omega \Vdash g), [\neg e], \Omega \Vdash g)}\ \text{SUFFICES}_2.$$

⟨3⟩1. $(\Gamma, \langle n\rangle l \triangleq (\Omega \Vdash g), [\neg e], [\langle n\rangle l] \Vdash e)$ is provable — *By* i.h. for $\mathcal{D}_0$, weakening (Fact 9).
⟨3⟩2. $(\Gamma, \langle n\rangle l \triangleq (\Omega \Vdash g), [\neg e], [(\Omega \Vdash g)] \Vdash e)$ is provable — *By* ⟨3⟩1, expansion (Fact 10).
⟨3⟩3. $(\Gamma, \langle n\rangle l \triangleq (\Omega \Vdash g), [\neg e] \Vdash e)$ is provable
By ⟨3⟩2, provability of $(\Gamma, \langle n\rangle l \triangleq (\Omega \Vdash g), [\neg e], \Omega \Vdash g)$, cut (Fact 12).
⟨3⟩4. $(\Gamma, [\neg e] \Vdash e)$ is provable — *By* ⟨3⟩3, strengthening (Fact 11).
⟨3⟩5. *Qed* — *By* ⟨3⟩4, Fact 13.

⟨2⟩20. *Case* τ is PICK $\vec{\beta} : p$ PROOF π, *i.e.*,

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{D}_0\\ \pi : \left(\Gamma \Vdash \exists \vec{\beta} : p\right)\end{array}}{\sigma.\ \text{PICK}\ \vec{\beta} : p\ \text{PROOF}\ \pi : (\Gamma \Vdash e) \longrightarrow \left(\Gamma, \left\|\vec{\beta}\right\|, p \Vdash e\right)}\ \text{PICK}.$$

⟨3⟩1. $\left(\Gamma, \exists \vec{\beta} : p \Vdash e\right)$ is provable — *By* provability of $\left(\Gamma, \left\|\vec{\beta}\right\|, p \Vdash e\right)$, predicate logic.
⟨3⟩2. $\left(\Gamma \Vdash \exists \vec{\beta} : p\right)$ is provable — *By* i.h. for $\mathcal{D}_0$.
⟨3⟩3. *Qed* — *By* ⟨3⟩1, ⟨3⟩2, cut (Fact 12).

⟨2⟩21. *Qed* — *By* ⟨2⟩1, ..., ⟨2⟩20
⟨1⟩3. *Qed* — *By* ⟨1⟩1, ⟨1⟩2.

□

A.4 Constrained Search

The correctness theorem (22) establishes an implication from the leaf obligations generated by a complete meaningful claim to the obligation of the claim. It is always true, regardless of the provability of any individual leaf obligation. While changing the visibility of assumptions in an obligation does not change its provability, a back-end prover may fail to prove it if important assumptions are hidden. As already mentioned in Section 3, the PM removes these hidden assumptions before sending a leaf obligation to a back-end prover. Therefore, in order to establish the Structural Soundness Theorem (1), we must prove a property about the result of this removal.

Definition 23 (Filtration)**.** *The* filtered *form of any obligation ϕ, written $(\phi)_{\mathrm{f}}$, is obtained by deleting all assumptions of the form $[\phi_0]$ and replacing all assumptions of the form $[o \triangleq \delta]$ with* NEW *o anywhere inside ϕ.*

For example, $(\text{NEW}\, x, [y \triangleq x] \Vdash x = y)_{\mathrm{f}} = (\text{NEW}\, x, \text{NEW}\, y \Vdash x = y)$. We thus see that filtration can render a true obligation false; however, if the filtered form of an obligation is true, then so is the obligation.

Lemma 24 (Verification Lemma)**.** *If $(\phi)_{\mathrm{f}}$ is provable, then ϕ is provable.*

Proof Sketch. By induction on the structure of the obligation ϕ, with each case a straightforward consequence of facts 8 and 9. □

Definition 25 (Verifiability)**.** *The obligation ϕ is said to be* verifiable *if $(\phi)_{\mathrm{f}}$ is provable.*

We now prove the Structural Soundness Theorem (1).

Theorem 1. *If $\pi : \phi$ is a complete meaningful claim and every leaf obligations it generates is verifiable, then ϕ is true.*

Proof.

⟨1⟩1. For every leaf obligation ϕ_0 generated by $\pi : \phi$, it must be that ϕ_0 is provable.

⟨2⟩1. *Take* ϕ_0 as a leaf obligation generated by $\pi : \phi$.

⟨2⟩2. $(\phi_0)_{\mathrm{f}}$ is provable — *By* assumption and Definition 25.

⟨2⟩3. *Qed* — *By* ⟨2⟩2, Verification Lemma 24.

⟨1⟩2. ϕ is provable — *By* ⟨1⟩1, Correctness Theorem 22.

⟨1⟩3. *Qed* — *By* ⟨1⟩2, Trust Axiom 7.

□

B A TLA^{+2} Proof of Cantor's Theorem

The following is the complete TLA^{+2} proof of Cantor's theorem referenced in Section 2.3.

THEOREM $\forall S : \forall f \in [S \to \text{SUBSET } S] : \exists A \in \text{SUBSET } S : \forall x \in S : f[x] \neq A$
⟨1⟩1. ASSUME NEW S,
NEW $f \in [S \to \text{SUBSET } S]$
PROVE $\exists A \in \text{SUBSET } S : \forall x \in S : f[x] \neq A$
⟨2⟩1. DEFINE $T \triangleq \{z \in S : z \notin f[z]\}$
⟨2⟩2. $\forall x \in S : f[x] \neq T$
⟨3⟩1. ASSUME NEW $x \in S$ PROVE $f[x] \neq T$
⟨4⟩1. CASE $x \in T$ OBVIOUS
⟨4⟩2. CASE $x \notin T$ OBVIOUS
⟨4⟩3. QED BY ⟨4⟩1, ⟨4⟩2
⟨3⟩2. QED BY ⟨3⟩1
⟨2⟩3. QED BY ⟨2⟩2
⟨1⟩2. QED BY ⟨1⟩1

As an example, the leaf obligation generated (see Appendix A.3) for the proof of ⟨4⟩1 is:

$\big($ ⟨1⟩1 $\triangleq$ (NEW S, NEW f, $f \in [S \to \text{SUBSET } S] \Vdash \exists A \in \text{SUBSET } S : \forall x \in S : f[x] \neq A$),
NEW S,
NEW f, $f \in [S \to \text{SUBSET } S]$,
$T \triangleq \{z \in S : z \notin f[z]\}$,
$[\neg(\exists A \in \text{SUBSET } S : \forall x \in S : f[x] \neq A)]$,
⟨2⟩2 $\triangleq \forall x \in S : f[x] \neq T$,
$[\neg(\forall x \in S : f[x] \neq T)]$,
⟨3⟩1 $\triangleq$ (NEW x, $x \in S \Vdash f[x] \neq T$),
NEW x, $x \in S$,
$[\neg(f[x] \neq T)]$,
⟨4⟩1 $\triangleq$ ($x \in T \Vdash f[x] \neq T$),
$x \in T$
$\Vdash f[x] \neq T$ $\big)$.

Filtering its obligation (see Definition 23) and expanding all definitions gives:

$\big($ NEW S,
NEW f, $f \in [S \to \text{SUBSET } S]$,
NEW x, $x \in S$,
$x \in \{z \in S : z \notin f[z]\} \Vdash f[x] \neq \{z \in S : z \notin f[z]\}$ $\big)$.

In Isabelle/TLA$^+$, this is the following lemma:

```
lemma ⋀S.
      ⋀f. f ∈ [S → SUBSET S] ⟹
          (⋀x. ⟦ x ∈ S;
                 x ∈ {z ∈ S : z ∉ f[z]} ⟧ ⟹ f[x] ≠ {z ∈ S : z ∉ f[z]})
```

The SZS Ontologies for Automated Reasoning Software

Geoff Sutcliffe
University of Miami

Abstract

This paper describes the SZS ontologies that provide status values for precisely describing what is known or has been established about logical data. The ontology values are useful for describing existing logical data, and for automated reasoning software to describe their input and output. Standards for presenting the ontology values are also provided.

1 Introduction

The real use of automated reasoning software - automated theorem proving (ATP) systems and other tools - is not as standalone software that a user invokes directly, but rather as embedded components of more complex reasoning systems. For one example, NASA's certifiable program synthesis system [6] embeds the SSCPA ATP system harness [20], the ATP systems E [13], SPASS [24], Vampire [12], and the GDV derivation verifier [17]. For another example, SRI's BioDeducta system [14] embeds the ATP system SNARK [15], and the BioBike integrated knowledge base and biocomputing platform [10]. In this embedded context automated reasoning software is typically treated as a black box with known processing capabilities. In order to use the software, the host system must know how to invoke the software, how to pass data into the software, and how to accept data produced by the software.

The data passed in to and out from automated reasoning software typically consists of *logical data*, e.g., formulae, derivations, interpretations, etc., and *status values* that describe what is known or has been established about the logical data, e.g., the nature of the formulae, their theoremhood or satisfiability, a reason why the software could not process the data, etc. For software that works with first-order logic, the de facto standard for expressing logical data is the TPTP language [19] (and it is expected that this will soon extend to higher-order logic [4]). The SZS ontologies that are linked to the TPTP are used by some automated reasoning software to express the status values. This paper describes the SZS ontologies and their use by automated reasoning software.

The status information output by current automated reasoning software varies widely in quantity, quality, and meaning. At the low end of the scale, for example, an ATP system might report only an assurance that the input problem's conjecture is a theorem of the axioms (the wonderful "`yes`" output). In some cases the claimed status is misleading, e.g., when a clause normal form refutation based ATP system claims that a first-order input problem consisting of axioms and a conjecture is "unsatisfiable", it typically means that the conjecture is a theorem of the axioms. At the high end of the scale, for example, a tool such as Infinox might report that a set of formulae does not have a finite model, or, for another example, a set of formulae might be tagged as representing a Herbrand interpretation. In order to seamlessly embed automated reasoning software in more complex reasoning systems, it is necessary to correctly and precisely specify status values for the input and output data. The SZS ontologies provide fine grained ontologies of status values that are suitable for this task.

The SZS *success ontology* provides status values to describe what is known or has been successfully established about the relationship between the axioms and conjecture in logical data. It is described in Section 2. The SZS *no-success ontology* provides status values to describe why a success ontology value has not been established. It is described in Section 3. The SZS *dataform ontology* provides status values

Rudnicki P, Sutcliffe G., Konev B., Schmidt R., Schulz S. (eds.);
Proceedings of the Combined KEAPPA - IWIL Workshops, pp. 38-49

to describe the nature of logical data. It is described in Section 4. All status values are expressed as "OneWord" to make system output parsing simple, and also have a three letter mnemonic. In addition to the ontologies themselves, standards for *presenting status values* have been specified. These are described in Section 5.

2 The SZS Success Ontology

The SZS success ontology was inspired by work done to establish communication protocols for systems on the MathWeb Software Bus [2, 25]. The ontology assumes that the logical data is a 2-tuple of the form $\langle Ax, C \rangle$, where Ax is a set (conjunction) of axioms and C is a conjecture formula. This is a common standard usage of ATP systems. If the input is not of the form $\langle Ax, C \rangle$, it is treated as a conjecture formula (even if it is a "set of axioms" from the user view point, e.g., a set of formulae all with the TPTP role `axiom`), and the 2-tuple is $\langle TRUE, C \rangle$. The success ontology values are based on the possible relationships between the sets of models of Ax and C. The ontology values can also be interpreted in terms of the formula $F \equiv Ax \Rightarrow C$. For example, the status `Theorem` means that the set of models of Ax is a (not necessarily strict) subset of the set of models of C, i.e., every model of Ax is a model of C. In this case F is valid.

Figure 1 shows the success ontology (many of the "OneWord" status values are abbreviated in the figure - see the list below for the official full "OneWord"s). The lines in the ontology can be followed up the hierarchy as *isa* links, e.g., an ETH *isa* EQV *isa* (SAT and a THM). Figure 2 shows the relationships between the model sets for some of the success ontology values. The outer grey ring contains all interpretations, the long dashed black ring contains the models of Ax, and the short dashed black ring contains the models of C.

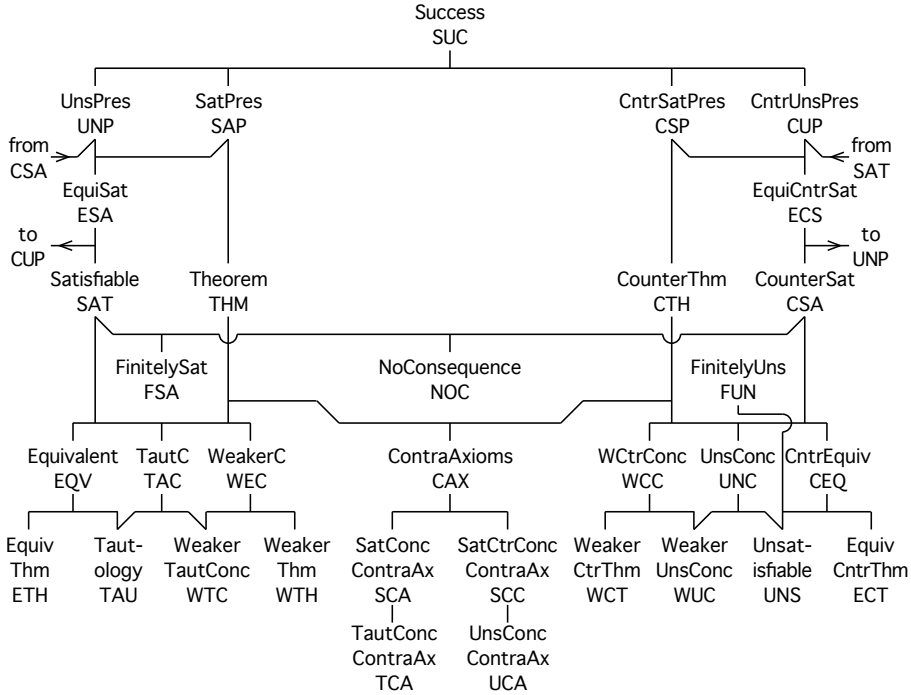


Figure 1: SZS Success Ontology

The meanings of the success ontology values are as follows. Associated with each status value are some possible dataforms that might be provided to justify the ontology value for given logical data - see

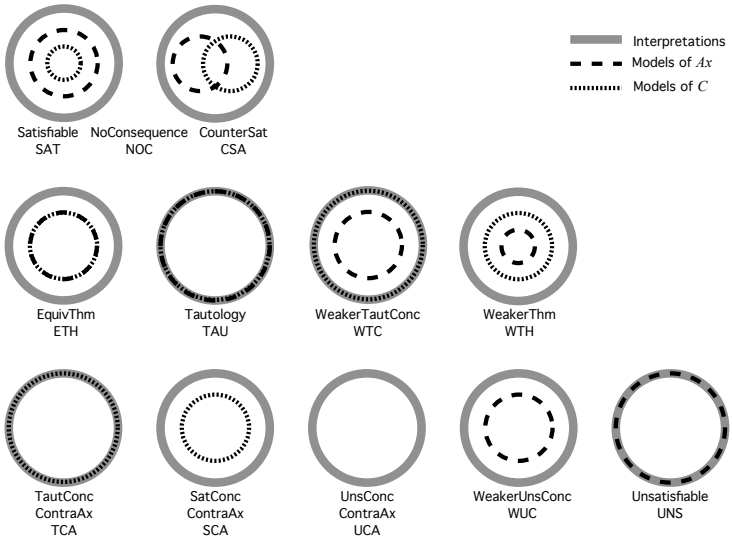


Figure 2: SZS Success Ontology Models

Section 4.

- Success (SUC): The logical data has been processed successfully.
- UnsatisfiabilityPreserving (UNP): If there does not exist a model of *Ax* then there does not exist a model of *C*, i.e., if *Ax* is unsatisfiable then *C* is unsatisfiable.
- SatisfiabilityPreserving (SAP): If there exists a model of *Ax* then there exists a model of *C*, i.e., if *Ax* is satisfiable then *C* is satisfiable. *F* is satisfiable.
- EquiSatisfiable (ESA): There exists a model of *Ax* iff there exists a model of *C*, i.e., *Ax* is (un)satisfiable iff *C* is (un)satisfiable.
- Satisfiable (SAT): Some interpretations are models of Ax, and some models of *Ax* are models of *C*. *F* is satisfiable, and $\neg F$ is not valid. Possible dataforms are Models of $Ax \wedge C$.
- FinitelySatisfiable (FSA): Some finite interpretations are finite models of *Ax*, and some finite models of *Ax* are finite models of *C*. *F* is satisfiable, and $\neg F$ is not valid. Possible dataforms are FiniteModels of $Ax \wedge C$.
- Theorem (THM): All models of *Ax* are models of *C*. *F* is valid, and *C* is a theorem of *Ax*. Possible dataforms are Proofs of *C* from *Ax*.
- Equivalent (EQV): Some interpretations are models of *Ax*, all models of *Ax* are models of *C*, and all models of *C* are models of *Ax*. *F* is valid, *C* is a theorem of *Ax*, and *Ax* is a theorem of *C*. Possible dataforms are Proofs of *C* from *Ax* and of *Ax* from *C*.
- TautologousConclusion (TAC): Some interpretations are models of *Ax*, and all interpretations are models of *C*. *F* is valid, and *C* is a tautology. Possible dataforms are Proofs of *C*.
- WeakerConclusion (WEC): Some interpretations are models of *Ax*, all models of *Ax* are models of *C*, and some models of *C* are not models of *Ax*. See Theorem and Satisfiable.
- EquivalentTheorem (ETH): Some, but not all, interpretations are models of *Ax*, all models of *Ax* are models of *C*, and all models of *C* are models of *Ax*. See Equivalent.
- Tautology (TAU): All interpretations are models of *Ax*, and all interpretations are models of *C*. *F* is valid, $\neg F$ is unsatisfiable, and *C* is a tautology. Possible dataforms are Proofs of *Ax* and of *C*.
- WeakerTautologousConclusion (WTC): Some, but not all, interpretations are models of *Ax*, and all interpretations are models of *C*. *F* is valid, and *C* is a tautology. See TautologousConclusion

and WeakerConclusion.

- WeakerTheorem (WTH): Some interpretations are models of Ax, all models of Ax are models of C, some models of C are not models of Ax, and some interpretations are not models of C. See Theorem and Satisfiable.
- ContradictoryAxioms (CAX): No interpretations are models of Ax. F is valid, and anything is a theorem of Ax. Possible dataforms are Refutations of Ax.
- SatisfiableConclusionContradictoryAxioms (SCA): No interpretations are models of Ax, and some interpretations are models of C. See ContradictoryAxioms.
- TautologousConclusionContradictoryAxioms (TCA): No interpretations are models of Ax, and all interpretations are models of C. See TautologousConclusion and ContradictoryAxioms.
- CounterUnsatisfiabilityPreserving (CUP): If there does not exist a model of Ax then there does not exist a model of $\neg C$, i.e., if Ax is unsatisfiable then $\neg C$ is unsatisfiable.
- CounterSatisfiabilityPreserving (CSP): If there exists a model of Ax then there exists a model of $\neg C$, i.e., if Ax is satisfiable then $\neg C$ is satisfiable.
- EquiCounterSatisfiable (ECS): There exists a model of Ax iff there exists a model of $\neg C$, i.e., Ax is (un)satisfiable iff $\neg C$ is (un)satisfiable.
- CounterSatisfiable (CSA): Some interpretations are models of Ax, and some models of Ax are models of $\neg C$. F is not valid, $\neg F$ is satisfiable, and C is not a theorem of Ax. Possible dataforms are Models of $Ax \wedge \neg C$.
- CounterTheorem (CTH): All models of Ax are models of $\neg C$. F is not valid, and $\neg C$ is a theorem of Ax. Possible dataforms are Proofs of $\neg C$ from Ax.
- CounterEquivalent (CEQ): Some interpretations are models of Ax, all models of Ax are models of $\neg C$, and all models of $\neg C$ are models of Ax. F is not valid, and $\neg C$ is a theorem of Ax. All interpretations are models of Ax xor of C. Possible dataforms are Proofs of $\neg C$ from Ax and of Ax from $\neg C$.
- UnsatisfiableConclusion (UNC): Some interpretations are models of Ax, and all interpretations are models of $\neg C$ (i.e., no interpretations are models of C). F is not valid, and $\neg C$ is a tautology. Possible dataforms are Proofs of $\neg C$.
- WeakerCounterConclusion (WCC): Some interpretations are models of Ax, and all models of Ax are models of $\neg C$, and some models of $\neg C$ are not models of Ax. See CounterTheorem and CounterSatisfiable.
- EquivalentCounterTheorem (ECT): Some, but not all, interpretations are models of Ax, all models of Ax are models of $\neg C$, and all models of $\neg C$ are models of Ax. See CounterEquivalent.
- FinitelyUnsatisfiable (FUN): All finite interpretations are finite models of Ax, and all finite interpretations are finite models of $\neg C$ (i.e., no finite interpretations are finite models of C).
- Unsatisfiable (UNS): All interpretations are models of Ax, and all interpretations are models of $\neg C$. (i.e., no interpretations are models of C). F is unsatisfiable, $\neg F$ is valid, and $\neg C$ is a tautology. Possible dataforms are Proofs of Ax and of C, and Refutations of F.
- WeakerUnsatisfiableConclusion (WUC): Some, but not all, interpretations are models of Ax, and all interpretations are models of $\neg C$. See Unsatisfiable and WeakerCounterConclusion.
- WeakerCounterTheorem (WCT): Some interpretations are models of Ax, all models of Ax are models of $\neg C$, some models of $\neg C$ are not models of Ax, and some interpretations are not models of $\neg C$. See CounterSatisfiable.
- SatisfiableCounterConclusionContradictoryAxioms (SCC): No interpretations are models of Ax, and some interpretations are models of $\neg C$. See ContradictoryAxioms.
- UnsatisfiableConclusionContradictoryAxioms (UCA): No interpretations are models of Ax, and all

interpretations are models of $\neg C$ (i.e., no interpretations are models of C). See UnsatisfiableConclusion and ContradictoryAxioms.
- NoConsequence (NOC): Some interpretations are models of Ax, some models of Ax are models of C, and some models of Ax are models of $\neg C$. F is not valid, F is satisfiable, $\neg F$ is not valid, $\neg F$ is satisfiable, and C is not a theorem of Ax. Possible dataforms are pairs of models, one Model of $Ax \wedge C$ and one Model of $Ax \wedge \neg C$.

The success ontology is very fine grained, and has more status values than are commonly used by automated reasoning software, by ATP systems in particular. A suitable subset for practical uses of ATP systems is as follows:

- FOF problems with a conjecture - report Theorem or CounterSatisfiable.
- FOF problems without a conjecture - report Satisfiable or Unsatisfiable.
- CNF problems - report Satisfiable or Unsatisfiable.

2.1 Validation of the Success Ontology

Two steps have been taken towards formal validation of the success ontology. The first step was the enumeration of the possible relationships between the models of Ax and C (some of which are illustrated in Figure 2). This provided a basis for the ontology values, and a basis for the *isa* links. The second step[1] was to axiomatize the ontology and prove relevant properties. (The axiomatization implemented covers the "positive" part of the ontology regarding Ax and C, and just two commonly used values from the "negative" part regarding Ax and$\neg C$. It is expected that the results obtained will extend without difficulty to the full ontology.) The axiomatization encodes the relationship between the models of Ax and C for each ontology value, and, from that, relationships between the ontology values can be proven. Additionally, a finite model of the axioms was found, demonstrating the consistency of the axiomatization and hence the ontology.

The axiomatization is in first-order logic. As example, the axioms that describe the ESA, THM, and ETH values are given in Figure 3. Four relationships between pairs of ontology values were defined and axiomatized:

- α *isa* β, meaning that if $\langle Ax, C\rangle$ has the status α then it also has the status β. For example, WTH *isa* THM.
- α *nota* β, meaning that if $\langle Ax, C\rangle$ has the status α then it does not necessarily have the status β. For example, THM *nota* SAT (because SAT does not hold for the case of contradictory Ax).
- α *nevera* β, meaning that if $\langle Ax, C\rangle$ has the status α then it cannot have the status β. For example, SAT *nevera* CAX.
- α *xora* β, meaning that every $\langle Ax, C\rangle$ has the status α xor β. For example, THM *xora* CSA.

Additionally, axioms that deal with properties of formulae and models were provided. The relationships and properties axioms are given in Figure 3.

The axiomatization was shown to be consistent by generating a finite model using Paradox [5]. Some general properties of the relationships were proved using an ATP system (see below for a discussion of the ATP system used), e.g., that *isa* is a transitive relation, and that if α *isa* β and α *nota* γ then β *nota* γ. Next the relationships between all pairs of ontology values were investigated, using the ATP system to prove the relationships from the axioms. The *isa* relationship was tested first, as if a pair of ontology values has the *isa* relationship they cannot have any of the other three relationships. For those pairs that were not proved to have the *isa* relationship, the *nevera* relationship was tested next. For those pairs that

[1] Thanks to the reviewer of this paper whose comments instigated this step.

```
fof(esa,axiom,(
    ! [Ax,C] :
      ( ( ? [I1] : model(I1,Ax)
      <=> ? [I2] : model(I2,C) )
    <=> status(Ax,C,esa) ) )).

fof(thm,axiom,(
    ! [Ax,C] :
      ( ! [I1] :
          ( model(I1,Ax)
         => model(I1,C) )
    <=> status(Ax,C,thm) ) )).

fof(eth,axiom,(
    ! [Ax,C] :
      ( ( ? [I1] : model(I1,Ax)
        & ? [I2] : ~ model(I2,Ax)
        & ! [I1] :
            ( model(I1,Ax)
            <=> model(I1,C) ) )
      <=> status(Ax,C,eth) ) )).

fof(isa,axiom,(
    ! [S1,S2] :
      ( ! [Ax,C] :
          ( status(Ax,C,S1)
         => status(Ax,C,S2) )
    <=> isa(S1,S2) ) )).

fof(nota,axiom,(
    ! [S1,S2] :
      ( ? [Ax,C] :
          ( status(Ax,C,S1)
          & ~ status(Ax,C,S2) )
    <=> nota(S1,S2) ) )).

fof(nevera,axiom,(
    ! [S1,S2] :
      ( ! [Ax,C] :
          ( status(Ax,C,S1)
         => ~ status(Ax,C,S2) )
    <=> nevera(S1,S2) ) )).

fof(xora,axiom,(
    ! [S1,S2] :
      ( ! [Ax,C] :
          ( status(Ax,C,S1)
        <~> status(Ax,C,S2) )
    <=> xora(S1,S2) ) )).

fof(completeness,axiom,(
    ! [I,F] :
      ( model(I,F)
    <~> model(I,not(F)) ) )).

fof(not,axiom,(
    ! [I,F] :
      ( model(I,F)
    <=> ~ model(I,not(F)) ) )).

fof(tautology,axiom,(
    ? [F] : ! [I] : model(I,F) )).

fof(contradiction,axiom,(
    ? [F] : ! [I] : ~ model(I,F) )).

fof(sat_non_taut_pair,axiom,(
    ? [Ax,C] :
      ( ? [I1] :
          ( model(I1,Ax)
          & model(I1,C) )
      & ? [I2] :
          ( ~ model(I2,Ax)
          | ~ model(I2,C) ) ) )).

fof(satisfiable,axiom,(
    ? [F] :
      ( ? [I1] : model(I1,F)
      & ? [I2] : ~ model(I2,F) ) )).

fof(non_thm_spt,axiom,(
    ? [I1,Ax,C] :
      ( model(I1,Ax)
      & ~ model(I1,C)
      & ? [I2] : model(I2,C) ) )).
```

Figure 3: SZS Success Ontology Axioms

were proved to have the *nevera* relationship the *xora* relationship was tested, and for the other pairs the *nota* relationship was tested. Proving a *nota* relationship requires establishing the existence of formulae and models that deny the relationship. In the axiomatization five examples are provided, the `tautology`, `satisfiable`, `contradiction`, `non_thm_spt` and `sat_non_taut_pair` axioms above.

The results of the testing are shown in Table 1, where the vertical axis value has the shown relationship to the horizontal axis value. The *isa* relationship is denoted by $\Rightarrow$, *nota* by $\neg$, *nevera* by $\times$, and *xora* by $\oplus$. Sixty-four pairs were proved to have the *isa* relationship, 85 to have the *nota* relationship, 151 to have the *nevera* (and not the *xora*) relationship, 4 to have the *xora* relationship, and for the remaining two pairs no relationship could be proved. The latter are the cases that WEC *nota* WTC and WEC *nota* TAC, which require exhibition of an $\langle Ax, C \rangle$ pair that has the WEC property but in which C is not a tautology. This could be done explicitly, along the lines of the `sat_non_taut_pair` axiom above, but that seemed like cheating. Some of the *nota* relationships may also be *nevera*, but could not be proved so.

As mentioned above, automated theorem proving was used to prove the relationships between the ontology values. At first, proofs were attempted using monolithic ATP systems such as EP, SPASS, and Vampire. The success rate was low, because the axiomatization forms a large theory - see [1]. Therefore the SRASS system [18] was used, and it was highly successful in identifying the necessary axioms for proving each conjecture, and subsequently obtaining either a proof using EP or an assurance of a proof using iProver [9]. In addition to SRASS, the MANSEX [22] and IDV [23] tools were used during the initial development of the axiomatization, to find the most obvious relationships and to analyze proofs. All automated reasoning and proof processing was done on a computer with a Intel Xeon 2.80GHz CPU and 3GB memory, running the Linux 2.6 operating system, and with a 60s CPU time limit per proof attempt (on the entire SRASS process).

	UNP	SAP	ESA	SAT	THM	EQV	TAC	WEC	ETH	TAU	WTC	WTH	CAX	SCA	TCA	CSA	UNS	NOC
UNP	●	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬	⊕	×	¬	¬	¬
SAP	¬	●	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬	×	¬
ESA	⇒	⇒	●	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬	×	×	¬	×	¬
SAT	⇒	⇒	⇒	●	¬	¬	¬	¬	¬	¬	¬	¬	×	×	×	¬	×	¬
THM	¬	⇒	¬	¬	●	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬	⊕	×	×
EQV	⇒	⇒	⇒	⇒	⇒	●	¬	×	¬	¬	×	×	×	×	×	×	×	×
TAC	⇒	⇒	⇒	⇒	⇒	¬	●	¬	×	¬	¬	×	×	×	×	×	×	×
WEC	⇒	⇒	⇒	⇒	⇒	×	¬	●	×	×	¬	¬	×	×	×	×	×	×
ETH	⇒	⇒	⇒	⇒	⇒	⇒	×	×	●	×	×	×	×	×	×	×	×	×
TAU	⇒	⇒	⇒	⇒	⇒	⇒	⇒	×	×	●	×	×	×	×	×	×	×	×
WTC	⇒	⇒	⇒	⇒	⇒	×	⇒	⇒	×	×	●	×	×	×	×	×	×	×
WTH	⇒	⇒	⇒	⇒	⇒	×	×	⇒	×	×	×	●	×	×	×	×	×	×
CAX	¬	⇒	¬	×	⇒	×	×	×	×	×	×	×	●	¬	¬	×	×	×
SCA	⊕	⇒	×	×	⇒	×	×	×	×	×	×	×	⇒	●	¬	×	×	×
TCA	×	⇒	×	×	⇒	¬	×	×	×	×	×	×	⇒	⇒	●	×	×	×
CSA	⇒	¬	¬	¬	⊕	×	×	×	×	×	×	×	×	×	×	●	¬	¬
UNS	⇒	×	×	×	×	×	×	×	×	×	×	×	×	×	×	⇒	●	×
NOC	⇒	⇒	⇒	⇒	×	×	×	×	×	×	×	×	×	×	×	⇒	×	●

Table 1: Success Ontology Relationships

The formal analysis has had beneficial effects. Two new ontology values were added, three errors in the definitions of the ontology values were exposed and corrected, four incorrect *isa* links in the ontology were found and removed, and several unnoticed *isa* relationships were revealed and added. The *isa* links in Figure 1 correspond to those in Table 1.

3 The SZS No-Success Ontology

While it is always hoped that automated reasoning software will successfully process the logical data, and hence establish a success ontology value, in reality this often does not happen, for a variety of reasons. In order to understand and make productive use of a lack of success, e.g., [11, 8], it is necessary to precisely specify the reason for and nature of the lack of success. The SZS no-success ontology provides suitable status values for describing the reasons. Note that no-success is not the same as failure: failure means that the software has completed its attempt to process the logical data and could not establish a success ontology value. In contrast, no-success might be because the software is still running, or that it has not yet even started processing the logical data. Figure 4 shows the no-success ontology.

The meanings of the no-success ontology values are as follows:

- NoSuccess (NOS): The logical data has not been processed successfully (yet).
- Open (OPN): A success value has never been established.
- Unknown (UNK): Success value unknown, and no assumption has been made.
- Assumed (ASS(U,S)): The success ontology value S has been assumed because the actual value is unknown for the no-success ontology reason U. U is taken from the subontology starting at Unknown in the no-success ontology.
- Stopped (STP): Software attempted to process the data, and stopped without a success status.
- Error (ERR): Software stopped due to an error.

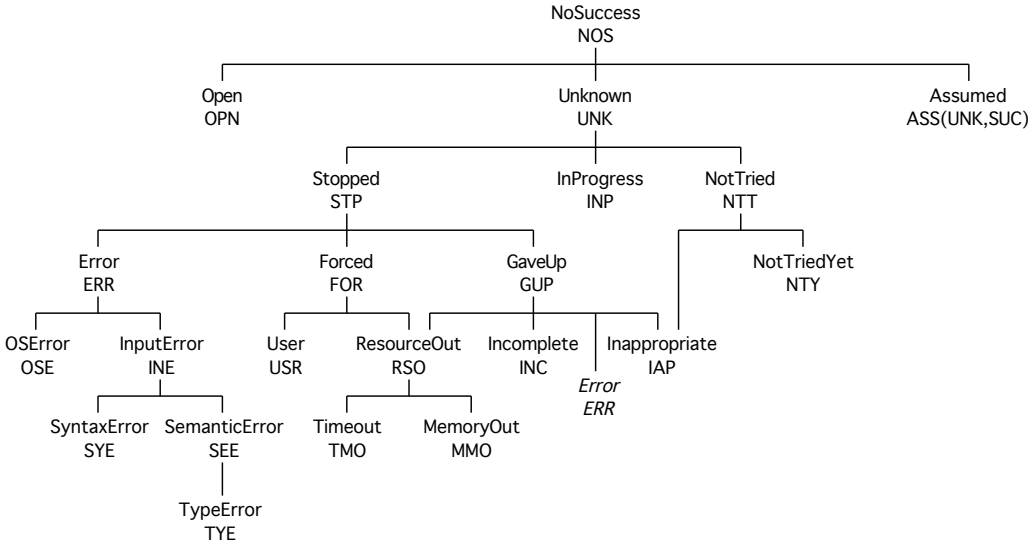


Figure 4: SZS No-Success Ontology

- OSError (OSE): Software stopped due to an operating system error.
- InputError (INE): Software stopped due to an input error.
- SyntaxError (SYE): Software stopped due to an input syntax error.
- SemanticError (SEE): Software stopped due to an input semantic error.
- TypeError (TYE): Software stopped due to an input type error (for typed logical data, e.g., in THF).
- Forced (FOR): Software was forced to stop by an external force.
- User (USR): Software was forced to stop by the user.
- ResourceOut (RSO): Software stopped because some resource ran out.
- Timeout (TMO): Software stopped because the CPU time limit ran out.
- MemoryOut (MMO): Software stopped because the memory limit ran out.
- GaveUp (GUP): Software gave up of its own accord.
- Incomplete (INC): Software gave up because it's incomplete.
- Inappropriate (IAP): Software gave up because it cannot process this type of data.
- InProgress (INP): Software is still running.
- NotTried (NTT): Software has not tried to process the data.
- NotTriedYet (NTY): Software has not tried to process the data yet, but might in the future.

The no-success ontology is very fine grained, and has more status values than are commonly used by automated reasoning software. A suitable subset for practical uses is as follows:

- The software stopped due to CPU limit - report Timeout.
- The software gave up due to incompleteness - report GaveUp.
- The software stopped due to an error - report Error.
- Any other cases - report Unknown.

4 The SZS Dataform Ontology

The success status values describe what is known or has been established about the relationship between the axioms and conjecture in logical data, but do not describe the form of logical data. The dataform ontology provides suitable values for describing the form of logical data. The dataform ontology values are commonly used to describe data provided to justify a success ontology value, e.g., if an ATP system reports the success ontology value `Theorem` it might output a proof to justify that. Figure 5 shows the dataform ontology.

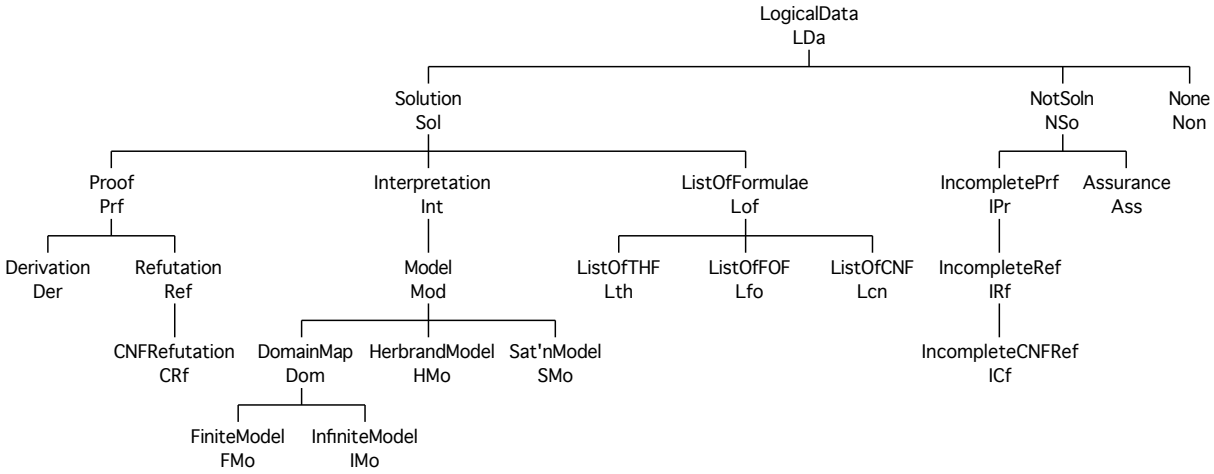


Figure 5: SZS Dataform Ontology

The meanings of the dataform ontology values are as follows:

- LogicalData (LDa): Logical data.
- Solution (Sln): A solution.
- Proof (Prf): A proof.
- Derivation (Der): A derivation (inference steps ending in the theorem, in the Hilbert style).
- Refutation (Ref): A refutation (starting with $Ax \cup \neg C$ and ending in FALSE).
- CNFRefutation (CRf): A refutation in clause normal form, including, for FOF Ax or C, the translation from FOF to CNF (without the FOF to CNF translation it is an IncompleteRefutation).
- Interpretation (Int): An interpretation.
- Model (Mod): A model.
- DomainMap (Dom): A model expressed as a domain, a mapping for functions, and a relation for predicates.
- FiniteModel (FMo): A domain map model with a finite domain.
- InfiniteModel (IMo): A domain map model with an infinite domain.
- HerbrandModel (HMo): A model expressed as a subset of the Herbrand base.
- SaturationModel (SMo): A model expressed as a saturating set of formulae.
- ListofFormulae (Lof): A list of formulae.
- ListofTHF (Lth): A list of THF formulae.
- ListofFOF (Lfo): A list of FOF formulae.
- ListofCNF (Lcn): A list of CNF formulae.
- IncompleteProof (IPr): A proof with part missing.
- IncompleteRefutation (IRf): A refutation with parts missing.

- IncompleteCNFRefutation (ICf): A CNF refutation with parts missing.
- Assurance (Ass): Only an assurance of the success ontology value.
- None (Non): Nothing.

The dataform ontology is very fine grained, and has more status values than are commonly used by automated reasoning software, by ATP systems in particular. A suitable subset for practical uses of ATP systems is as follows:

- A generic proof - report Proof.
- A refutation - report Refutation.
- A CNF refutation - report CNFRefutation.
- A generic model - report Model.
- A finite model - report FiniteModel.
- A Herbrand model - report HerbrandModel.
- A saturation model - report SaturationModel.

5 The SZS Presentation Standards

The SZS ontologies provide status values that precisely describe what is known or has been established about logical data. In order to make the use of the values easy in more complex reasoning systems, it is necessary to specify precisely how the values should be presented. This makes it easy for harness software to prepare input data and examine output data that contains ontology values, e.g., in practice, to `grep` the output from automated reasoning software for lines that provide status values.

Success and no-success ontology values should be presented in lines of the form

`% SZS status` *ontology_value* `for` *logical_data_identifier*

(The leading '%' makes the line into a TPTP language comment.) For example

```
% SZS status Unsatisfiable for SYN075+1
```

or

```
% SZS status GaveUp for SYN075+1
```

A success or no-success ontology value should be presented as early as possible, at least before any data output to justify the value. The justifying data should be delimited by lines of the form

`% SZS output start` *dataform_ontology_value* `for` *logical_data_identifier*

and

`% SZS output end` *dataform_ontology_value* `for` *logical_data_identifier*

For example

```
% SZS output start CNFRefutation for SYN075-1
    output_data
% SZS output end CNFRefutation for SYN075-1
```

All "SZS" lines can optionally have software specific information appended, separated by a `:`, i.e.,

`% SZS status` *ontology_value* `for` *logical_data_identifier* `:` *software_specific_information*

`% SZS output start` *dataform_ontology_value* `for` *logical_data_identifier* `:` *software_specific_info*

`% SZS output end` *dataform_ontology_value* `for` *logical_data_identifier* `:` *software_specific_info*

For example

```
% SZS status GaveUp for SYN075+1 : Could not complete CNF conversion
```

or

```
% SZS output end CNFRefutation for SYN075-1 : Completed in CNF conversion
```

6 Conclusion

This paper has presented the SZS ontologies of status values that are suitable for expressing precisely what is known or has been established about logical data. The ontologies can be used for existing logical data, e.g., they are used for the status of problems in the TPTP problem library [21] and solutions in the TSTP solution library [16], and can be used by automated reasoning software to describe their input and output. Already several ATP systems, e.g., Darwin [3], E, Metis [7], Paradox [5], use the SZS ontologies and the presentation standards, and this contributes to simplifying their embedding into more complex reasoning systems.

In addition to its use for reporting the overall status of a $\langle Ax, C \rangle$ 2-tuple, the SZS success ontology is used to report the status of individual inference steps in TPTP format derivations [19]. This is done in the "useful information" field of an `inference` record of an inferred formula. For example, in

```
cnf(58,plain,
    ( ~ hates(agatha,esk2_1(butler)) ),
    inference(spm,[status(thm)],[51,48])).
```

the status is Theorem (recorded as a lowercase acronym value `thm`), which indicates that the formulae is a theorem of it's two parent formulae `51` and `48`. The Theorem status is most common in derivations, but the SAP and ESA status values are also used quite often, e.g., for the formulae inferred by Skolemization and splitting steps. These status values can be used for semantic verification of the derivations, as is done by the GDV derivation verifier [17].

While the SZS ontologies are in use and have matured to some extent, it is not claimed that they are comprehensive and perfect. Developers and users of automated reasoning software are invited to provide feedback that might lead to improvements and increased usage. Already some users are working on success ontology values for results from computer algebra and other computational reasoning systems. In related work, SZS standards for returning answers from question-and-answer systems have been proposed.[2] It is hoped that over time, with increased usage, the ontologies will become battle hardened, and will be a core standard for automated reasoning.

References

[1] *Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories*, number 257 in CEUR Workshop Proceedings, 2007.

[2] A. Armando, M. Kohlhase, and S. Ranise. Communication Protocols for Mathematical Services based on KQML and OMRS. In M. Kerber and M. Kohlhase, editors, *Proceedings of the Calculemus Symposium 2000*, 2000.

[3] 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.

[4] C. Benzmüller, F. Rabe, and G. Sutcliffe. THF0 - The Core TPTP Language for Classical Higher-Order Logic. In P. Baumgartner, A. Armando, and D. Gilles, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, page Accepted, 2008.

[5] K. Claessen and N. Sorensson. 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.

[6] E. Denney, B. Fischer, and J. Schumann. Using Automated Theorem Provers to Certify Auto-generated Aerospace Software. In M. Rusinowitch and D. Basin, editors, *Proceedings of the 2nd International Joint*

[2] See `http://www.tptp.org/TPTP/Proposals/AnswerExtraction.html`

Conference on Automated Reasoning, number 3097 in Lecture Notes in Artificial Intelligence, pages 198–212, 2004.

[7] J. Hurd. First-Order Proof Tactics in Higher-Order Logic Theorem Provers. In M. Archer, B. Di Vito, and C. Munoz, editors, *Proceedings of the 1st International Workshop on Design and Application of Strategies/-Tactics in Higher Order Logics*, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, 2003.

[8] A. Ireland and A. Bundy. Productive use of Failure in Inductive Proof. *Journal of Automated Reasoning*, 16(1-2):79–111, 1996.

[9] K. Korovin. iProver - An Instantiation-Based Theorem Prover for First-order Logic (System Description). In P. Baumgartner, A. Armando, and D. Gilles, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, number 5195 in Lecture Notes in Artificial Intelligence, pages 292–298, 2008.

[10] P. Massar, M. Travers, J. Elhai, and J. Shrager. BioLingua: A Programmable Knowledge Environment for Biologists. *Bioinformatics*, 21(2):199–207, 2005.

[11] R. Monroy, A. Bundy, and A. Ireland. Proof Plans for the Correction of False Conjectures. In F. Pfenning, editor, *Proceedings of the 5th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 822 in Lecture Notes in Artificial Intelligence, pages 178–189. Springer-Verlag, 1994.

[12] A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.

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

[14] J. Shrager, R. Waldinger, M. Stickel, and P. Massar. Deductive Biocomputing. *PLoS ONE*, 2(4), 2007.

[15] M.E. Stickel. SNARK - SRI's New Automated Reasoning Kit. http://www.ai.sri.com/ stickel/snark.html.

[16] G. Sutcliffe. The TSTP Solution Library. http://www.TPTP.org/TSTP.

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

[18] 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.

[19] 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.

[20] G. Sutcliffe and D. Seyfang. Smart Selective Competition Parallelism ATP. In A. Kumar and I. Russell, editors, *Proceedings of the 12th International FLAIRS Conference*, pages 341–345. AAAI Press, 1999.

[21] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[22] G. Sutcliffe, A. Yerikalapudi, and S. Trac. Multiple Answer Extraction for Question Answering with Automated Theorem Proving Systems. Rejected from the 15th International Conference on Logic for Programming Artificial Intelligence and Reasoning, 2008.

[23] 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.

[24] C. Weidenbach, R. Schmidt, T. Hillenbrand, R. Rusev, and D. Topic. SPASS Version 3.0. In F. Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction*, number 4603 in Lecture Notes in Artificial Intelligence, pages 514–520. Springer-Verlag, 2007.

[25] J. Zimmer and M. Kohlhase. System Description: The MathWeb Software Bus for Distributed Mathematical Reasoning. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, number 2392 in Lecture Notes in Artificial Intelligence, pages 139–143. Springer-Verlag, 2002.

An Exchange Format for Modular Knowledge

Florian Rabe, Michael Kohlhase
Jacobs University Bremen

Abstract

We present a knowledge exchange format that is designed to support the exchange of modularly structured information in mathematical and logical systems. This format allows to encode mathematical knowledge in a logic-neutral representation format that can represent the meta-theoretic foundations of the systems together with the knowledge itself and to interlink the foundations at the meta-logical level. This "logics-as-theories" approach, makes system behaviors as well as their represented knowledge interoperable and thus comparable. The advanced modularization features of our system allow communication between systems without losing structure.

We equip the proposed format with a web-scalable XML/URI-based concrete syntax so that it can be used as a universal interchange and archiving format for existing systems.

1 Introduction

Mathematical knowledge is at the core of science, engineering, and economics, and we are seeing a trend towards employing computational systems like semi-automated theorem provers, model checkers, computer algebra systems, constraint solvers, or concept classifiers to deal with it. It is a characteristic feature of these systems that they either have mathematical knowledge implicitly encoded in their critical algorithms or (increasingly) manipulate explicit representations of the relevant mathematical knowledge often in the form of logical formulae. Unfortunately, these systems have differing domains of applications, foundational assumptions, and input languages, which makes them non-interoperable and difficult to compare and evaluate in practice. Moreover, the quantity of mathematical knowledge is growing faster than our ability to formalize and organize it, aggravating the problem that mathematical software systems cannot share knowledge representations.

The work reported in this paper focuses on developing an exchange format between math-knowledge based systems. We concentrate on a foundationally unconstrained framework for knowledge representation that allows to represent the meta-theoretic foundations of the mathematical knowledge in the same format and to interlink the foundations at the meta-logical level. In particular, the logical foundations of domain representations for the mathematical knowledge can be represented as modules themselves and can be interlinked via meta-morphisms. This "logics-as-theories" approach, makes systems behavior as well as their represented knowledge interoperable and thus comparable at multiple levels. Note that the explicit representation of epistemic foundations also benefits systems whose mathematical knowledge is only implicitly embedded into the algorithms. Here, the explicit representation can serve as a documentation of the system as well as a basis for verification or testing attempts.

Of course communication by translation to the lowest common denominator logic — the current state of the art — is always possible. But such translations lose the very structural properties of the knowledge representation that drive computation and led to the choice of logical system in the first place. Therefore our format incorporates a module system geared to support flexible reuse of knowledge items via theory morphisms. This module system is the central part of the proposed format — emphasizing interoperability between theorem proving systems, and the exchange and reusability of mathematical facts across different systems. In contrast to the formula level, which needs to be ontologically unconstrained, the module system level must have a clear semantics (relative to the semantics of the formula level) and

Rudnicki P, Sutcliffe G., Konev B., Schmidt R., Schulz S. (eds.);
Proceedings of the Combined KEAPPA - IWIL Workshops, pp. 50-66

be expressive enough to encode current structuring practice so that systems can communicate without losing representational structure.

On a practical level, an exchange format should support interoperability by being easy to parse and standards-compatible. Today, where XML-parsers are available for almost all programming language and higher-level protocols for distributed computation are XML-based, an exchange format should be XML-based and in particular an extension of the MATHML [ABC+03] and OPENMATH [BCC+04] standards. Moreover, an exchange format must be *web-scalable* in the sense that it supports the distribution of resources (theories, conjectures, proofs, etc.) over the Internet, so that they can be managed collaboratively. In the current web architecture this means that all (relevant) resources must be addressable by a URI-based naming scheme [BLFM05]. Note that in the presence of complex modularity and reusability infrastructure this may mean that resources have to be addressable, even though they are only virtually induced by the inheritance structure.

2 Syntax

2.1 A Four-Level Model of Mathematical Knowledge

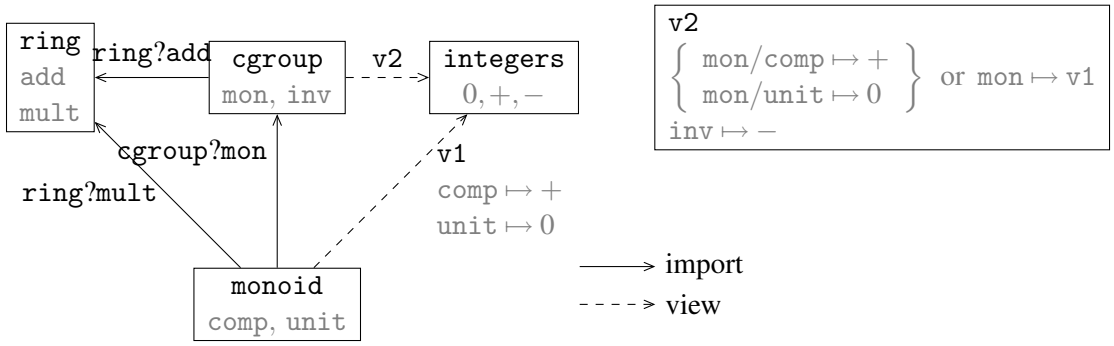


Figure 1: Example

Example 1. We begin the exposition of our language with a simple motivating example that already shows some of the crucial features of MMT and that we will use as a running example: Fig. 1 gives a theory graph for a portion of the algebraic hierarchy. The bottom node represents the theory of monoids, which declares operations for composition and unit. (We omit the axioms and the types here.) The theory `cgroup` for commutative groups arises by importing from `monoid` and adding an operation `inv` for the inverse element. This theory does not have to declare the operations for composition and unit again because they are imported from the theory of monoids. The import is named `mon`, and it can be referenced by the qualified name `cgroup?mon`; it induces a theory morphism from monoids to commutative groups.

Then the theory of rings can be formed by importing from `monoid` (via an import named `mult` that provides the multiplicative structure) and from `cgroup` (via an import named `add` that provides the additive structure). Because all imports are named, different import paths can be distinguished: By concatenating import and symbol names, the theory `ring` can access the symbols `add/mon/comp` (i.e., addition), `add/mon/unit` (i.e, zero), `add/inv` (i.e., additive inverse), `mult/comp` (i.e., multiplication), and `mult/unit` (i.e., one).

The node on the right side of the graph represents a theory for the integers declaring the operations 0, $+$, and $-$. The fact that the integers are a monoid is represented by the view `v1`. It is a theory morphism that is explicitly given by its interpretations of `comp` as $+$ and of `unit` as 0. (If we did not omit axioms,

this view would also have to interpret all the axioms of `monoid` as — using Curry-Howard representation — proof terms.)

The view `v2` is particularly interesting because there are two ways to represent the fact that the integers are a commutative group. Firstly, all operations of `cgroup` can be interpreted as terms over `integers`: This means to interpret `inv` as $-$ and the two imported operations `mon/comp` and `mon/unit` as $+$ and 0, respectively. Secondly, `v2` can be constructed along the modular structure of `cgroup` and use the existing view `v1` to interpret all operations imported by `mon`. In MMT, this can be expressed elegantly by the interpretation `mon` $\mapsto$ `v1`, which interprets a named import with a theory morphism. The intuition behind such an interpretation is that it makes the right triangle commute: `v2` is defined such that `v2` $\circ$ *cgroup*?*mon* = `v1`. Clearly, both ways lead to the same theory morphism; the second one is conceptually more complex but eliminates redundancy. (This redundancy is especially harmful when axioms are considered, which must be interpreted as proofs.)

Ontology and Grammar We characterize mathematical theories on four levels: the **document**, **module**, **symbol**, and **object** level. On each level, there are several kinds of expressions. In Fig. 2, the relations between the MMT-concepts of the first three levels are defined in an ontology. The MMT knowledge items are grouped into six primitive concepts. **Documents** (*Doc*, e.g., the whole graph in Fig. 1) comprise the document level. **Theories** (*Thy*, e.g., `monoid` and `integers`) and **views** (*Viw*, e.g., `v1` and `v2`) comprise the module level. And finally **constants** (*Con*, e.g., `comp` and `inv`) and **structures** (*Str*, e.g., `mon` and `add`) as well as **assignments** to them (*ConAss*, e.g., `inv` $\mapsto -$, and *StrAss*, e.g., `mon` $\mapsto$ `v2`) comprise the symbol level.

In addition, we define four unions of concepts: First **modules** (*Mod*) unite theories and views, **symbols** (*Sym*) unite constants and structures, and **assignments** (*Ass*) unite the assignments to constants and structures. The most interesting union is that of **links** (*Lnk*): They unite the module level concept of views and the symbol level concept of structures. Structures being both symbols and links makes our approach balanced between the two different ways to understand them.

These higher three levels are called the structural levels because they represent the structure of mathematical knowledge: Documents are sets of modules, theories are sets of symbols, and links are sets of assignments. The actual mathematical objects are represented at the fourth level: They occur as arguments of symbols and assignments. MMT provides a formalization of the structural levels while being parametric in the specific choice of objects used on the fourth level.

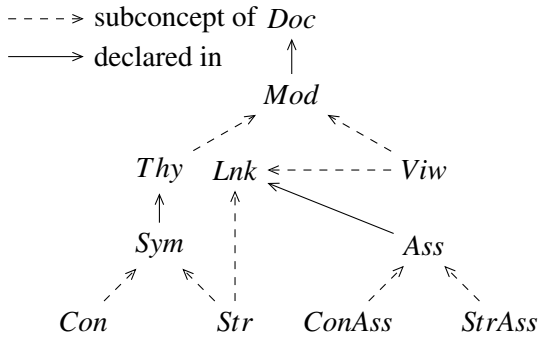


Figure 2: The MMT Ontology

The declarations of the four levels along with the meta-variables we will use to reference them are given in Fig. 3 and 4. The MMT knowledge items of the structural levels are declared with unqualified names (underlined Latin letter) in a certain scope and referenced by qualified names (Latin letter). Greek letters are used as meta-variables for composed expressions. Following practice in programming languages, we will use _ as an unnamed meta-variable for irrelevant values.

The grammar for MMT is given in Fig. 5 where *, $^+$, $|$, and $[-]$ denote repetition, non-empty repetition, alternative, and optional parts, respectively. The rules for the non-terminals URI and pchar are defined in RFC 3986. Thus, *g* produces a URI without a query or a fragment. (The query and fragment components of a URI are those starting with the special characters ? and #, respectively.) pchar, essentially, produces any Unicode character, possibly using percent-encoding for reserved characters. (Thus,

Level	Declared concept
Document	document g
Module	theory T, S, R, M
	view v
	structure or view m
Symbol	symbol s
	constant c
	structure h, i, j
	assignment to a constant
	assignment to a structure
Object	variable x

Figure 3: Meta-Variables for References to MMT Declarations

Variable	Type
Λ	library (list of document declarations)
γ	document body (list of module declarations)
ϑ	theory body (list of symbol declarations)
σ	view/structure body (list of assignments to symbols)
ω	term
μ	morphism

Figure 4: Meta-Variables for MMT Expressions

percent-encoding is necessary for the characters ?/#[]% and all characters generally illegal in URIs.) In this section, we will describe the syntax of MMT and the intuition behind it in a bottom-up manner, i.e., from the object to the document level. Alternatively, the following subsections can be read in top-down order.

2.1.1 The Object Level

We distinguish two kinds of objects: **terms** and **morphisms**. Atomic terms are references to declared constants, and general terms are built up from the constants via operations such as application and binding as detailed below. Atomic morphisms are references to structures and views, and general morphisms are built up using structures, views, identity, and composition as detailed below.

The MMT objects are typed. For terms, we do not distinguish terms and types syntactically: Rather, typing is a binary relation on terms that is not further specified by MMT. Instead, it is left to be determined by the **foundation**, and MMT is parametric in the specific choice of typing relation. In particular, terms may be untyped or may have multiple types. For morphisms, the domain theory doubles as the type. Thus, every morphism has a unique type.

Well-formedness of objects is checked relative to a home theory. For a term ω, the home theory must declare or import all symbols that occur in ω. For a morphism, the home theory is the codomain. Objects with home theory T are also called **objects over** T. These relations are listed in Fig. 6.

Terms MMT-terms are a generalization of a fragment of OPENMATH objects ([BCC+04]). They can be

Library	Λ	::=	Doc^*
Document	Doc	::=	$g := \{\gamma\}$
Document body	γ	::=	Mod^*
Module	Mod	::=	$Thy \mid Viw$
Theory	Thy	::=	$\underline{T} \overset{[T]}{:=} \{\vartheta\}$
View	Viw	::=	$\underline{v} : T \to T \overset{[\mu]}{:=} \{\sigma\} \mid \underline{v} : T \to T := \mu$
Theory body	ϑ	::=	Sym^*
Symbol	Sym	::=	$Con \mid Str$
Link body	σ	::=	Ass^*
Assignment	Ass	::=	$ConAss \mid StrAss$
Constant	Con	::=	$\underline{c} : \omega := \omega \mid \underline{c} : \omega \mid \underline{c} := \omega \mid \underline{c}$
Structure	Str	::=	$\underline{i} : T \overset{[\mu]}{:=} \{\sigma\} \mid \underline{i} : T := \mu$
Ass. to constant	$ConAss$	::=	$\underline{c} \mapsto \omega$
Ass. to structure	$StrAss$	::=	$\underline{i} \mapsto \mu$
Term	ω	::=	$\top \mid c \mid x \mid \omega^{\mu} \mid @(\omega, \omega^+)$ $\mid \beta(\omega, \Upsilon, \omega) \mid \alpha(\omega, \omega \mapsto \omega)$
Variable context	Υ	::=	$\cdot \mid \Upsilon, \omega$ if $str(\omega)$ of the form x
Morphism	μ	::=	$id_T \mid i \mid v \mid \mu \bullet \mu$
Document reference	g	::=	URI, no query, no fragment
Module reference	T, v	::=	$g?\underline{T} \mid g?\underline{v}$
Symbol reference	c, i	::=	$T?\underline{c} \mid T?\underline{i}$
Assignment reference	a	::=	$v?\underline{c} \mid v?\underline{i}$
Local name	$\underline{T}, \underline{v}, \underline{c}, \underline{i}$	::=	$C^+[/C^+]^*$
Variable name	x	::=	C^+
Character	C	::=	pchar
	URI, pchar		see RFC 3986 [BLFM05]

Figure 5: The Grammar for Raw MMT Expressions

	Atomic object	Composed object	Type	Checked relative to
Terms	constant	term	term	home theory
Morphisms	structure/view	morphism	domain	codomain

Figure 6: The Object Level

- **constants** c declared or imported by the home theory,
- **variables** x declared by a binder,
- **applications** $@(\omega, \omega_1, \ldots, \omega_n)$ of ω to arguments ω_i,
- **bindings** $\beta(\omega_1, \Upsilon, \omega_2)$ by a binder ω_1 of a list of variables Υ with scope ω_2,
- **attributions** $\alpha(\omega_1, \omega_2 \mapsto \omega_3)$ to a term ω_1 with key ω_2 and value ω_3,
- **morphism applications** ω^{μ} of μ to ω,

- **special term** $\top$.

A term over T, may use the constant $T?\underline{c}$ to refer to a previously declared symbol $\underline{c}$. And if $\underline{i}$ is a previously declared structure instantiating S, and $\underline{c}$ is a constant declared in S, then T may use $T?\underline{i}/\underline{c}$ to refer to the imported constant. By concatenating structure names, any indirectly imported constant has a unique qualified name.

The attributions of OPENMATH mean that every term can carry a list of key-value pairs that are themselves OPENMATH objects. In particular, attributions are used to attach types to bound variables. In OPENMATH, the keys must be symbols, which we relax to obtain a more uniform syntax. Because OPENMATH specifies that nested attributions are equivalent to a single attribution, we can introduce attributions of multiple key-value pairs as abbreviations:

$$\alpha(\omega, \omega_1 \mapsto \omega_1', \ldots, \omega_n \mapsto \omega_n') := \alpha(\ldots(\alpha(\omega, \omega_1 \mapsto \omega_1'), \ldots), \omega_n \mapsto \omega_n')$$

We use the auxiliary function $str(\cdot)$ to strip toplevel attributions from terms, i.e.,

$$str(\alpha(\omega, \_)) = str(\omega) \qquad str(\omega) = \omega \text{ otherwise.}$$

This is used in the grammar to make sure that only attributed variables and not arbitrary terms may occurs in the context bound in a binding.

The special term $\top$ is used for terms that are inaccessible because they refer to constants that have been hidden by a structure or view. $\top$ is also used to subsume hidings under assignments (see below).

Example 2 (Continued). The running example only contains the atomic terms given by symbols. Composed terms arise when types and axioms are covered. For example, the type of the inverse in a commutative group is $@(\rightarrow, \iota, \iota)$. Here $\rightarrow$ represents the function type constructor and ι the carrier set. These two constants are not declared in the example. Instead, we will add them later by giving `cgroup` a meta-theory, in which these symbols are declared. A more complicated term is the axiom for left-neutrality of the unit:

$$\omega_e := \beta(\forall, \alpha(x, \texttt{oftype} \mapsto \iota), @(=, @(e?\texttt{monoid}?\texttt{comp}, e?\texttt{monoid}?\texttt{unit}, x), x)).$$

Here $\forall$ and $=$ are further constants that must be declared in the so far omitted meta-theory. The same applies to `oftype`, which is used to attribute the type ι to the bound variable x. We assume that the example is located in a document with URI e. Thus, for example, $e?\texttt{monoid}?\texttt{comp}$ is used to refer to the constant `comp` in the theory `monoid`.

Morphisms Morphisms are built up by compositions $\mu_1 \bullet \mu_2$ of structures i, views m, and the identity id_T of T. Here μ_1 is applied before μ_2, i.e., $\bullet$ is composition in diagram order. Morphisms are not used in OPENMATH, which only has an informal notion of theories, namely the content dictionaries, and thus no need to talk about theory morphisms. A morphism application ω^μ takes a term ω over S and a morphism μ from S to T, and returns a term over T. Similarly, a morphism over S, e.g., a morphism μ' from R to S becomes a morphism over T by taking the composition $\mu' \bullet \mu$. The normalization given below will permit to eliminate all morphism applications.

Example 3 (Continued). In the running example, an example morphism is

$$\mu_e := e?\texttt{cgroup}?\texttt{mon} \bullet e?\texttt{v2}.$$

It has domain $e?\texttt{monoid}$ and codomain $e?\texttt{integers}$. The intended semantics of the term ${\omega_e}^{\mu_e}$ is that it yields the result of applying μ_e to ω_e, i.e.,

$$\beta(\forall, \alpha(x, \texttt{oftype} \mapsto \iota), @(=, @(+, 0, x), x)).$$

Here, we assume μ_e has no effect on those constants that are inherited from the meta-theory. We will make that more precise below.

2.1.2 The Symbol Level

We distinguish symbol declarations and assignments to symbols. **Declarations** are the constituents of theories: They introduce named objects, i.e., the **constants** and **structures**. Similarly, **assignments** are the constituents of links: A link from S to T can be defined by a sequence of assignments that instantiate constants or structures declared in S with terms or morphisms, respectively, over T. This yields four kinds of knowledge items which are listed in Fig. 7. Both constant and structure declarations are further subdivided as explained below.

	Declaration	Assignment
Terms	of a constant *Con*	to a constant $\underline{c} \mapsto \omega$
Morphisms	of a structure *Str*	to a structure $\underline{i} \mapsto \mu$

Figure 7: The Statement Level

Declarations There are two kinds of symbols:

- **Constant declarations** $\underline{c} : \tau := \delta$ declare a constant $\underline{c}$ of type τ with definition δ. Both the type and the definition are optional yielding four kinds of constant declarations. If both are given, then δ must have type τ. In order to unify these four kinds, we will sometimes write $\bot$ for an omitted type or definition.

- **Structure declarations** $\underline{i} : S \stackrel{[\mu]}{:=} \{\sigma\}$ declare a structure $\underline{i}$ from the theory S defined by assignments σ. Such structures can have an optional meta-morphism μ (see below). Alternatively, structures may be introduced using an existing morphism: $\underline{i} : S := \mu$, which simply means that $\underline{i}$ serves as an abbreviation for μ; we call these structures **defined structures**. While the domain of a structure is always given explicitly (in the style of a type), the codomain is always the theory in which the structure is declared. Consequently, if $\underline{i} : S := \mu$ is declared in T, μ must be a morphism from S to T.

In well-formed theory bodies, the declared or imported names must be unique.

Assignments Parallel to the declarations, there are two kinds of assignments that can be used to define a link m:

- **Assignments to constants** of the form $\underline{c} \mapsto \omega$ express that m maps the constant $\underline{c}$ of S to the term ω over T. Assignments of the form $\underline{c} \mapsto \top$ express that the constant $\underline{c}$ is **hidden**, i.e., m is undefined for $\underline{c}$.

- **Assignments to structures** of the form $\underline{i} \mapsto \mu$ for a structure $\underline{i}$ declared in S and a morphism μ over (i.e., into) T express that m maps the structure $\underline{i}$ of S to μ. This results in the commuting triangle $S?\underline{i} \bullet m = \mu$.

Both kinds of assignments must type-check. For a link m with domain S and codomain T defined by among others an assignment $\underline{c} \mapsto \omega$, the term ω must type-check against τ^m where τ is the type of $\underline{c}$ declared in S. This ensures that typing is preserved along links. For an assignment $\underline{i} \mapsto \mu$ where $\underline{i}$ is a structure over S of type R, type-checking means that μ must be a morphism from R to T.

Virtual Symbols Intuitively, the semantics of a structure $\underline{i}$ with domain S declared in T is that all symbols of S are imported into T. For example, if S contains a symbol $\underline{s}$, then $\underline{i}/\underline{s}$ is available as a symbol in T. In other words, the slash is used as the operator that dereferences structures. Another way to say it is that structures create virtual or induced symbols. This is significant because these virtual symbols and their properties must be inferred, and this is non-trivial because it is subject to the translations given by the structure. While these induced symbols are easily known to systems built for a particular formalism, they present great difficulties for generic knowledge management services.

Similarly, every assignment to a structure induces virtual assignments to constants. Continuing the above example, if a link with domain T contains an assignment to $\underline{i}$, this induces assignments to the imported symbols $\underline{i}/\underline{s}$. Furthermore, assignments may be **deep** in the following sense: If $\underline{c}$ is a constant of S, a link with domain T may also contain assignments to the virtual constant $\underline{i}/\underline{c}$. Of course, this could lead to clashes if a link contains assignments for both $\underline{i}$ and $\underline{i}/\underline{c}$; links with such clashes are not well-formed.

Example 4 (Continued). The symbol declarations in the theory `cgroup` are written formally like this:

$$\texttt{inv} : @(\rightarrow, \iota, \iota) \quad \text{and} \quad \texttt{mon} : e?\texttt{monoid} := \{\}.$$

The latter induces the virtual symbols $e?\texttt{cgroup}?\texttt{mon}/\texttt{comp}$ and $e?\texttt{cgroup}?\texttt{mon}/\texttt{unit}$.

Using an assignment to a structure, the assignments of the view `v2` look like this:

$$\texttt{inv} \mapsto e?\texttt{integers}?- \quad \text{and} \quad \texttt{mon} \mapsto e?\texttt{v1}.$$

The latter induces virtual assignments for the symbols $e?\texttt{cgroup}?\texttt{mon}/\texttt{comp}$ and $e?\texttt{cgroup}?\texttt{mon}/\texttt{unit}$. For example, $e?\texttt{cgroup}?\texttt{mon}/\texttt{comp}$ is mapped to $e?\texttt{monoid}?\texttt{comp}^{e?\texttt{v1}}$.

The alternative formulation of the view `v2` arises if two deep assignments to the virtual constants are used instead of the assignment to the structure `mon`:

$$\texttt{mon}/\texttt{comp} \mapsto e?\texttt{integers}?+ \quad \text{and} \quad \texttt{mon}/\texttt{unit} \mapsto e?\texttt{integers}?0$$

2.1.3 The Module Level

The module level consists of two kinds of declarations: theory and view declarations.

- **Theory declarations** $\underline{T} \stackrel{[M]}{:=} \{\vartheta\}$ declare a theory $\underline{T}$ defined by a list of symbol declarations ϑ, which we call the body of $\underline{T}$. Theories have an optional meta-theory M.

- **View declarations** $\underline{\nu} : S \rightarrow T \stackrel{[\mu]}{:=} \{\sigma\}$ declare a link $\underline{\nu}$ from S to T defined by a list of assignments σ. If S has a meta-theory M, a meta-morphism μ from M to T must be provided. Just like structures, views may also be defined by an existing morphism: $\underline{\nu} : S \rightarrow T := \mu$.

Meta-Theories Above, we have already mentioned that theories may have meta-theories and that links may have meta-morphisms. Meta-theories provide a second dimension in the graph induced by theories and links. If M is the meta-theory of T, then there is a special structure instantiating M in T, which we denote by $T?...$ M provides the syntactic material that T can use to define the semantics of its symbols: T can refer to a symbol $\underline{s}$ of M by $T?../\underline{s}$. While meta-theories could in principle be replaced with structures altogether, it is advantageous to make them explicit because the conceptual distinction pervades mathematical discourse. For example, systems can use the meta-theory to determine whether they understand a specific theory they are provided as input.

Because theories S with meta-theory M implicitly import all symbols of M, a link from S to T must provide assignments for these symbols as well. This is the role of the meta-morphism: Every link from S to T must provide a meta-morphism, which must be a morphism from M to T. Defined structures or views with definition μ do not need a meta-morphism because a meta-morphism is already implied by the meta-morphisms of the links occurring in μ.

Example 5 (Continued). An MMT theory for the logical framework LF could be declared like this

$$\texttt{lf} := \{\texttt{type}, \texttt{funtype}, \ldots\}$$

where we only list the constants that are relevant for our running example. If this theory is located in a document with URI m, we can declare a theory for first-order logic in a document with URI f like this:

$$\texttt{fol} \overset{m?\texttt{lf}}{:=} \{\texttt{i} : ??../\texttt{type}, \texttt{o} : ??../\texttt{type}, \texttt{equal} : @(??../\texttt{funtype}, ??\texttt{i}, ??\texttt{i}, ??\texttt{o}), \ldots\}$$

Here we already use relative names (see Sect. 2.3.2) in order to keep the notation readable: Every name of the form $??\underline{s}$ is relative to the enclosing theory: For example, $??\texttt{i}$ resolves to $f?\texttt{fol}?\texttt{i}$. Furthermore, we use the special structure name .. to refer to those constants inherited from the meta-theory. Again we restrict ourselves to a few constant declarations: types `i` and `o` for terms and formulas and the equality operation that takes two terms and returns a formula.

Then the theories `monoid`, `cgroup`, and `ring` can be declared using $f?\texttt{fol}$ as their meta-theory. For example, the declaration of the theory `cgroup` finally looks like this:

$$\texttt{cgroup} \overset{f?\texttt{fol}}{:=} \left\{\texttt{inv} : @(??../../\texttt{funtype}, ??\texttt{i}, ??\texttt{i}), \texttt{inv} : e?\texttt{monoid} \overset{e?\texttt{cgroup}?..}{:=} \{\}\right\}$$

Here $../../\texttt{funtype}$ refers to the function type constant declared in the meta-theory of the meta-theory. And the structure `mon` must have a meta-morphism with domain $f?\texttt{fol}$ and codomain $e?\texttt{cgroup}$. This is trivial because the meta-theory of $e?\texttt{cgroup}$ is also $f?\texttt{fol}$: The meta-morphism is simply the implicit structure $e?\texttt{cgroup}?..$ via which $e?\texttt{cgroup}$ inherits from $f?\texttt{fol}$.

A more complicated meta-morphism must be given in the view `v1` if we assume that the meta-theory of `integers` is some other theory, i.e., a representation of set theory.

Structures and Views Both structures and views from S to T are defined by a list of assignments σ that assigns T-objects to the symbols declared in S. And both induce theory morphisms from S to T that permit to map all objects over S to objects over T. The major difference between structures and views is that a view only relates two fixed theories without changing either one. On the other hand, structures from S to T occur within T because they change the theory T. Structures have **definitional flavor**, i.e., the symbols of S are imported into T. In particular, if σ contains no assignment for a constant $\underline{c}$, this is equivalent to copying (and translating) the declaration of $\underline{c}$ from S to T. If σ does provide an assignment $\underline{c} \mapsto \omega$, the declaration is also copied, but in addition the imported constant receives ω as its definiens.

Views, on the other hand, have **theorem flavor**: σ *must* provide assignments for the symbols of S. If a constant $\underline{c} : \tau$ represents an axiom stating τ, giving an assignment $\underline{c} \mapsto \pi$ means that π is a proof of the translation of τ.

Therefore, the assignments defining a structure may be (and typically are) partial whereas a view should be total. This leads to a crucial technical difficulty in the treatment of structures: Contrary to views from S to T, the assignments by themselves in a structure from S to T do not induce a theory morphism from S to T — only by declaring the structure do the virtual symbols become available in T that serve as the images of (some of) the symbols of S. This is unsatisfactory because it makes it harder to unify the concepts of structures and views.

Therefore, we admit **partial views** as well. As it turns out, this is not only possible, but indeed desirable. A typical scenario when working with views is that some of the specific assignments making up the view constitute proof obligations and must be found by costly procedures. Therefore, it is reasonable to represent partial views, namely views where some proof obligations have already been discharged whereas others remain open. Thus, we use hiding to obtain a semantics for partial views: All constants for which a view does not provide an assignment are implicitly hidden, i.e., $\top$ is assigned to them.

If a link m from S to T is applied to an S-constant that is hidden, there are two cases: If the hidden symbol has a definition in S, it is replaced by this definition before applying the link. If it does not have a definition, it is mapped to $\top$. Hiding is strict: If a subterm is mapped to $\top$, then so is the whole term. In that case, we speak of hidden terms.

2.1.4 The Document Level

Document declarations are of the form $g := \{\gamma\}$ where γ is a document body and g is a URI identifying the document. The meaning of a document declaration is that γ is accessible via the name g. Since g is a URI, it is not necessarily only the name, but can also be the primary location of γ. By forming lists of documents, we obtain **libraries**, which represent mathematical knowledge bases. Special cases of libraries are single self-contained documents and the internet seen as a library of MMT documents.

Documents provide the border between formal and informal treatment: How documents are stored, copied, cached, mirrored, transferred, and combined is subject to knowledge management services that may or may not take the mathematical semantics of the document bodies in the processed documents into account. For example, libraries may be implemented as web servers, file systems, databases, or any combination of these. The only important thing is that they provide the query interface described below.

Theory graphs are the central notion of MMT. The theory graph is a directed acyclic multigraph. The nodes are all theories of all documents in the library. And similarly, the edges are the structures and views of all documents. Then theory morphisms can be recovered as the paths in the theory graph.

2.2 Querying a Library

MMT is designed to scale to a mathematical web. This means that we define access functions to MMT libraries that have the form of HTTP requests to a RESTful web server [Fie00]. Specifically, there is a lookup function that takes a library Λ and a URI U as arguments and returns an MMT fragment $\Lambda(U)$. This specifies the behavior of a web server hosting an MMT library in response to GET requests. Furthermore, all possible changes to an MMT library can be formulated as POST, PUT, and DELETE requests that add, change, and delete knowledge items, respectively.

It is non-trivial to add such a RESTful interface to formal systems a posteriori. It requires the rigorous use of URIs as identifiers for all knowledge items that can be the object of a change. And it requires to degrade gracefully if the documents in a library are not in the same main memory or on the same machine. In large applications, it is even desirable to load only the relevant parts of a document into the main memory and request further fragments on demand from a low-level database service. In MMT, web-scalability is built into the core: All operations on a library Λ including the definition of the semantics only depend on the lookup function $\Lambda(-)$ and not on Λ itself. In particular, two libraries that respond in the same way to lookup requests are indistinguishable by design. Therefore, MMT scales well to web-based scenarios.

Here we will only give a brief overview over the lookup function $\Lambda(-)$. It is a partial function from URIs to MMT fragments. For example, assume Λ to contain a theory S containing the symbol declarations $\underline{c} : \tau$, $\underline{c}' : \tau'$, and $\underline{h} : R := \{\}$; furthermore, assume a theory $T = g?\underline{T}$ declaring a structure $\underline{i} : S := \{\underline{c} \mapsto \delta, \underline{h} \mapsto \mu\}$.

Then the lookups of $S?\underline{c}$ and $S?\underline{h}$ yield $\underline{c} : \tau$ and $\underline{h} : R := \{\}$, respectively. These are lookups of explicit symbol declarations — an important feature of MMT is that all induced, symbols can be looked up in the same way. For example, $\Lambda(T?\underline{i}/\underline{c})$ yields $\underline{i}/\underline{c} : \tau^{T?\underline{i}} := \delta$. Here $\underline{i}/\underline{c}$ is the unique name of the constant $\underline{c}$ induced by the structure $\underline{i}$; $\tau^{T?\underline{i}}$ is the translation of the type of $S?\underline{c}$ along the structure $T?\underline{i}$; and the assignment $\underline{c} \mapsto \delta$ causes δ to occur as the definiens of the constant $\underline{i}/\underline{c}$. Similarly, the lookup of the induced structure $T?\underline{i}/\underline{h}$ yields $\underline{i}/\underline{h} : R := \mu$. Here the assignment $\underline{i} \mapsto \mu$ causes $\underline{i}/\underline{h}$ to be defined by a morphism.

To query and edit documents, assignments are also accessible by URIs even though they are not referenced by any rule of the syntax. An assignment $\underline{c} \mapsto \omega$ in a view v is referenced by the URI $v?\underline{c}$ and similarly for assignments to structures. To access assignments in structures, we distinguish the URIs $g?\underline{T}?\underline{i}/\underline{c}$ and $g?\underline{T}/\underline{i}?\underline{c}$: The former identifies the constant $\underline{c}$ that is induced by $\underline{i}$ as defined above; the latter identifies the assignment $\underline{c} \mapsto \delta$ to the symbol $\underline{c}$ in the structure $T?\underline{i}$. In particular, the lookup of the former always is always defined, while the lookup of the latter is undefined if no assignment is present.

Just like symbols induced by structures have URIs, so have assignments induced by assignments to structures. For example, if additionally R declares a constant $\underline{d} : \rho := \bot$, then the lookup of $g?\underline{T}/\underline{i}?\underline{h}/\underline{d}$ yields $\underline{h}/\underline{d} \mapsto (R?\underline{d})^{\mu}$. This means that the assignment $\underline{h} \mapsto \mu$ in $T?\underline{i}$ induces an assignment to $\underline{h}/\underline{d}$. The lookup of assignments to induced structures is defined accordingly.

The above lookup functions are actually modified to accommodate for hiding. If the lookup of a constant would yield $\underline{c} : \tau := \delta$ according to the above definition, but δ is (or simplifies to) $\top$, then the lookup is actually undefined.

Example 6 (Continued). If Λ is a library containing the three documents with URIs m, f, and e from the running example, we obtain the following lookup results:

- $\Lambda(e?\texttt{monoid}) = (f?\texttt{fol}, \vartheta)$ where ϑ contains the declarations for `comp` and `unit`,
- $\Lambda(e?\texttt{cgroup/mon}) = (e?\texttt{monoid}, e?\texttt{cgroup}, e?\texttt{cgroup}?.., \cdot)$,
  i.e., $e?\texttt{cgroup/mon}$ is a morphism from $e?\texttt{monoid}$ to $e?\texttt{cgroup}$ with meta-morphism $e?\texttt{cgroup}?..$, and without any assignments,
- $\Lambda^{e?\texttt{monoid}}(\texttt{unit}) = (e?\texttt{monoid}?../\texttt{i}, \bot)$,
  i.e., the theory `monoid` has a constant `unit` with type $e?\texttt{monoid}?../\texttt{i}$ and no definition,
- $\Lambda^{e?\texttt{cgroup}}(\texttt{mon/unit}) = (e?\texttt{monoid}?../\texttt{i}^{e?\texttt{cgroup}?\texttt{mon}}, \bot)$,
  i.e., the type of the virtual constant `mon/unit` arises by translating the type from the source theory along the importing structure,
- $\Lambda^{e?\texttt{cgroup/mon}}(\texttt{unit}) = \bot$,
  i.e., the lookup is undefined because the structure $e?\texttt{cgroup/mon}$ does not have an assignment for `unit`,
- the lookup $\Lambda^{e?\texttt{v2}}(\texttt{mon/unit})$ yields $e?\texttt{integers}?0$ if the variant with the deep assignment $\texttt{mon/unit} \mapsto e?\texttt{integers}?0$ is used to define `v2`, and $e?\texttt{monoid}?\texttt{unit}^{e?\texttt{v1}}$ if the variant with the structure assignment $\texttt{mon} \mapsto e?\texttt{v1}$ is used.

2.3 Concrete Syntax

2.3.1 XML-Encoding

The XML grammar mostly follows the abstract grammar. Documents are `omdoc` elements with a theory graph, i.e., `theory` and `view` elements as children. The children of theories are `constant` and `structure`. And the children of views and structures are `maps` (assignment to a constant or structure).

Both terms and morphisms are represented by OPENMATH elements. And all names of theories are URIs. The grammar does not account for the well-formedness of objects and names. In particular, well-formedness is not checked at this level.

Formally, we define an encoding function $E(-)$ that maps MMT-expressions to sequences of XML elements. The precise definition of $E(-)$ is given in Fig. 9 and 10 where we assume that the following namespace bindings are in effect:

xmlns="http://www.omdoc.org/ns/omdoc"
xmlns:om="http://www.openmath.org/OpenMath"

And we assume the following content dictionary with cdbase `http://cds.omdoc.org/omdoc/mmt.omdoc`, which is itself given as an OMDOC theory:

```
<omdoc>
  <theory name="mmt">
    <constant name="hidden"/>
    <constant name="identity"/>
    <constant name="composition"/>
  </theory>
</omdoc>
```

We abbreviate the `OMS` elements referring to these symbols by *OMDoc*(`identity`) etc.

The encoding of the structural levels is straightforward. The encoding of objects is a generalization of the XML encoding of OPENMATH objects. We use the `OMS` element of OPENMATH to refer to symbols, theories, views, and structures. The symbol with name *OMDoc*(`hidden`) is used to encode the special term $\top$. To encode morphisms, *OMDoc*(`identity`) and *OMDoc*(`composition`) are used. *OMDoc*(`identity`) takes a theory as an argument and returns a morphism. *OMDoc*(`composition`) takes a list of structures and views as arguments and returns their (left-associative) diagram order composition. Morphism application is encoded by reusing the `OMA` element from OPENMATH.

The encoding of names is giving separately in Fig. 8 on the right. There are two different ways to encode names. In `OMS` elements, triples $(g,\underline{m},\underline{s})$ of document, module, and symbol name are used, albeit with more fitting attribute names. These triples correspond to the $(cdbase, cd, name)$ triples of the OPENMATH standard ([BCC+04]). We also use them to refer to module level names by omitting the *name* attribute. When names occur in attribute values, their URIs are used.

`OMS`-triple	$E(g?\underline{m}?\underline{s})$	base="g" module="$\underline{m}$" name="$\underline{s}$"
	$E(g?\underline{m})$	base="g" module="$\underline{m}$"
URI	$E(g?\underline{m}?\underline{s})$	$g?\underline{m}?\underline{s}$
	$E(g?\underline{m})$	$g?\underline{m}$

Figure 8: XML Encoding of Names

2.3.2 Relative Names

In practice it is very inconvenient to always give qualified names. Therefore, we define relative references as a relaxation of the syntax that is elaborated into the official syntax.

A **relative reference** consists of three optional components: a document reference g, a module reference m and a symbol reference s. We write relative references as triples (g,m,s) where we write $\bot$ if a component is omitted. g must be a URI reference as defined in RFC 3986 ([BLFM05]) but without query or fragment. m and s must be unqualified names, i.e., slash-separated non-empty sequences of names. Furthermore, m ans s may optionally start with a slash, which is used to distinguish absolute module and symbol references from relative ones.

An **absolute reference**, which serves as the base of the resolution, is an MMT-name G, $G?\underline{M}$, or $G?\underline{M}?\underline{S}$. Then the resolution of relative references is a partial function that takes a relative reference

Library	$E(Doc_1,\ldots,Doc_n)$	$E(Doc_1)\ \ldots\ E(Doc_n)$
Document	$E(g := \{Mod_1,\ldots,Mod_n\})$	<**omdoc**>$E(Mod_1)\ldots E(Mod_n)$</**omdoc**>
Theory	$E(\underline{T} \overset{[M]}{:=} \{Sym_1,\ldots,Sym_n\})$	<**theory** name="$\underline{T}$" [metatheory="$E(M)$"]> $E(Sym_1)\ldots E(Sym_n)$ </**theory**>
View	$E(\underline{v} : S \to T \overset{[\mu]}{:=} \{\sigma\})$	<**view** name="$\underline{v}$" from="$E(S)$" to="$E(T)$"> [<**metamorphism**>$E(\mu)$</**metamorphism**>] $E(\sigma)$ </**view**>
	$E(\underline{i} : S \to T := \mu)$	<**view** name="$\underline{i}$" from="$E(S)$" to="$E(T)$"> <**definition**> <**OMOBJ**>$E(\mu)$</**OMOBJ**> </**definition**> </**view**>

Figure 9: XML Encoding of Document and Module Level

$R = (g,m,s)$ and an absolute reference B as input and returns an MMT-name $resolve(B,R)$ as output. It is defined as follows:

- If $g \neq \bot$, then possible starting slashes of m and s are ignored and
  - if $R = (g,m,s)$: $resolve(B,R) = (G+g)?m?s$,
  - if $R = (g,m,\bot)$: $resolve(B,R) = (G+g)?m$,
  - if $R = (g,\bot,\bot)$: $resolve(B,R) = G+g$,

  where $G+g$ denotes the resolution of the URI reference g relative to the URI G as defined in RFC 3986 ([BLFM05]).
- If $g = \bot$ and $m \neq \bot$, then a possible starting slash of s is ignored and
  - if $R = (\bot,m,s)$: $resolve(B,R) = G?M+m?s$,
  - if $R = (\bot,m,\bot)$: $resolve(B,R) = G?M+m$,

  where $M+m$ resolves m relative to M: If M is not defined or if m starts with a slash, $M+m$ is m with a possible starting slash removed; otherwise, it is M/m.
- If $g = m = \bot$ and M is defined, then $resolve(B,R) = G?M?S+s$, where $S+s$ is defined like $M+m$ above.
- $resolve(B,R)$ is undefined otherwise.

Relative references can also be encoded as URIs: The triple (g,m,s) is encoded as $g?m?s$. If components are omitted, they are encoded as the empty string. Trailing (but not leading) ? characters can be dropped. For example,

- $(g,m,\bot)$ is encoded as $g?m$,
- $(\bot,/m,s)$ is encoded as $?/m?s$,
- $(\bot,\bot,s)$ is encoded as $??s$,

This encoding can be parsed back uniquely by splitting a URI into up to three components around the separator ?.

Constant	$E(\underline{c} : [\tau] := [\delta])$	<**constant** name="$\underline{c}$"> [<**type**> <**OMOBJ**>$E(\tau)$</**OMOBJ**> </**type**>] [<**definition**> <**OMOBJ**>$E(\delta)$</**OMOBJ**> </**definition**>] </**constant**>
Structure	$E(\underline{i} : S \stackrel{[\mu]}{:=} \{\sigma\})$	<**structure** name="$\underline{i}$" from="$E(S)$"> [<**metamorphism**>$E(\mu)$</**metamorphism**>] $E(\sigma)$ </**structure**>
	$E(\underline{i} : S := \mu)$	<**structure** name="$\underline{i}$" from="$E(S)$"> <**definition**> <**OMOBJ**>$E(\mu)$</**OMOBJ**> </**definition**> </**structure**>
Assignments	$E(Ass_1, \ldots, Ass_n)$	$E(Ass_1) \ldots E(Ass_n)$
	$E(\underline{c} \mapsto \omega)$	<conass name="$E(\underline{c})$"> <**OMOBJ**>$E(\omega)$</**OMOBJ**> </conass>
	$E(\underline{i} \mapsto \mu)$	<strass name="$E(\underline{i})$"> <**OMOBJ**>$E(\mu)$</**OMOBJ**> </strass>
Term	$E(c)$	<**om:OMS** $E(c)$/>
	$E(x)$	<**om:OMV** name="x"/>
	$E(\top)$	*OMDoc*(`hidden`)
	$E(\omega^{\mu})$	<**om:OMA**>$E(\mu)$ $E(\omega)$</**om:OMA**>
	$E(@(\omega_1, \ldots, \omega_n))$	<**om:OMA**>$E(\omega_1)$ … $E(\omega_n)$</**om:OMA**>
	$E(\beta(\omega_1, x_1, \ldots, x_n, \omega_2))$	<**om:OMBIND**> $E(\omega_1)$ <**om:OMBVAR**> $E(x_1)$ … $E(x_n)$ </**om:OMBVAR**> $E(\omega_2)$ </**om:OMBIND**>
	$E(\alpha(\omega_1, \omega_2 \mapsto \omega_3))$	<**om:OMATTR**> <**om:OMATP**>$E(\omega_2)$ $E(\omega_3)$</**om:OMATP**> $E(\omega_1)$ </**om:OMATTR**>
Morphism	$E(id_T)$	<**om:OMA**> *OMDoc*(`identity`) <**om:OMS** $E(T)$/> </**om:OMA**>
	$E(\mu_1 \bullet \ldots \bullet \mu_n)$	<**om:OMA**> *OMDoc*(`composition`) $E(\mu_1) \ldots E(\mu_n)$ </**om:OMA**>
	$E(i)$	<**om:OMS** $E(i)$/>
	$E(v)$	<**om:OMS** $E(v)$/>

Figure 10: XML Encoding of Symbol and Object Level

3 Advanced Concepts

In this section, we give an overview over the most important advanced concepts related to MMT. Full definitions are given in [Rab08].

Well-formed Expressions The well-formedness of expressions is defined by an inference system about MMT fragments. This system guarantees in particular the uniqueness of all names. The most difficult of the well-formedness definition is to deal with assignments to structures: In general, an assignment $\underline{i} \mapsto \mu$ to a structure with domain S may cause inconsistencies if a constant $S?\underline{c}$ has a definition in S that differs from $(S?\underline{c})^{\mu}$. The definition of well-formedness makes sure that such inconsistencies are prevented. And it does so in a way that can be checked efficiently because the modular structure is exploited throughout the well-formedness checking.

MMT does not provide a definition of well-*typed* terms or of logical consequence. Rather, the MMT inference system is parametric in the judgments for equality and typing of terms. The definitions of these judgments must be provided externally. In our MMT implementation, these judgments are defined by plugins where the meta-theory of the home theory of a term ω determines which plugin is used to check the well-typedness of ω.

Semantics The representation of theory graphs is geared towards expressing mathematical knowledge with the least redundancy by using inheritance between theories. This style of writing mathematics has been cultivated by the Bourbaki group ([Bou74]) and lends itself well to a systematic development of theories. However, it is often necessary to eliminate the modular structure, for example when interfacing with a system that cannot understand it or to elaborate modular theories into a non-modular trusted core.

Therefore, we define a so-called **flattening** operation that eliminates all structures, meta-theories, and morphisms, and reduces theories to collections of constants, possibly with types and definitions, which conforms to the non-modular logical view of theories. For a given MMT-library Λ, we can view the flattening of Λ as its semantics, since flattening eliminates all specific MMT-representation infrastructure. Essentially, the flattening replaces all structures with the induced symbols and all assignments to structures with the induced assignments as described in Sect. 2.2. The crucial invariant of the flattening is that if a library is well-formed, then so is its flattening; and furthermore, a library and its flattening are indistinguishable by the lookup function $\Lambda(-)$. This guarantees that flattening can be performed transparently. In particular, the above-mentioned external definitions of typing and equality are only needed for the non-modular case.

Applications There are two main use cases for MMT: the use of MMT as a simple interface language between (i) two logical systems as well as between (ii) a logical system and a knowledge management service.

The first use case uses MMT as an interlingua for system interoperability. This works well because the choice of primitives in MMT is the result of a careful trade-off between simplicity and expressivity. Thus, MMT provides a simple abstraction over the most important language constructs that are common to systems maintaining logical libraries, while deliberately avoiding other features that would have increased the burden on systems reading MMT. Here the logics-as-theories approach provides an extremely helpful way to document the semantics of the logic-specific concepts and their counterparts in other logics; furthermore, systems can use the meta-theory relation to detect which theories they can interpret and how to interpret them.

By knowledge management services, we mean services that can be applied to logical knowledge but are not primarily logical in nature. Such services include user interaction, concurrent versioning, management of change, and search. Implementing such services often requires intricate details about the underlying system, thus severely limiting the set of potential developers. But in fact these services can often be developed independently of the logical systems and with relatively little logical expertise. Here MMT comes in as an interface language providing knowledge management services with exactly the information they need while hiding all logic- and system-specific details. Therefore, for example, the concrete syntax of MMT fully marks up the term structure, while the well-formedness definition of MMT does not take type-checking into account.

4 Conclusion

We have presented an exchange format for mathematical knowledge that supports system interoperability by providing an infrastructure for efficiently re-using representations of mathematical knowledge and for formalizing foundational assumptions and structures of the underlying logical systems themselves in a joint web-scalable representation infrastructure.

We consider it a major achievement of the work reported here that the MMT format integrates theoretical aspects like the syntax/semantics interface of the module systems or meta-logical aspects with practical considerations like achieving web-scalability via a URI-based referencing scheme that is well-integrated with the module system.

The proposed MMT format takes great care to find a minimal set of primitives to keep the representational core of the language small and therefore manageable but expressive enough to cover the semantic essentials of current (semi)-automated theorem provers, model checkers, computer algebra systems, constraint solvers, concept classifiers and mathematical knowledge management systems.

We are currently evaluating the MMT format and in particular are developing translators from the Isabelle [Pau94] and CASL [CoF04] formats into MMT. Both representation formats have strong module systems and well-specified underlying semantics, which make them prime evaluation targets. We are also integrating the MMT infrastructure into the upcoming version 2 of the OMDOC format [Koh06].

References

[ABC+03] Ron Ausbrooks, Stephen Buswell, David Carlisle, Stéphane Dalmas, Stan Devitt, Angel Diaz, Max Froumentin, Roger Hunter, Patrick Ion, Michael Kohlhase, Robert Miner, Nico Poppelier, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical Markup Language (MathML) version 2.0 (second edition). W3C recommendation, World Wide Web Consortium, 2003.

[BCC+04] S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See `http://www.openmath.org/standard/om20`.

[BLFM05] Tim Berners-Lee, Roy. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax. RFC 3986, Internt Engineering Task Force, 2005.

[Bou74] N. Bourbaki. *Algebra I*. Elements of Mathematics. Springer, 1974.

[CoF04] CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.

[Fie00] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[Koh06] Michael Kohlhase. OMDOC – *An open markup format for mathematical documents [Version 1.2]*. Number 4180 in LNAI. Springer Verlag, 2006.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS. Springer Verlag, 1994.

[Rab08] F. Rabe. *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen, 2008.

Connecting Gröbner bases programs with Coq to do proofs in algebra, geometry and arithmetics

Loïc Pottier
INRIA Sophia Antipolis

Abstract

We describe how we connected three programs that compute Gröbner bases [1] to Coq [11], to do automated proofs on algebraic, geometrical and arithmetical expressions. The result is a set of Coq tactics and a certificate mechanism [1]. The programs are: F4 [5], GB [4], and gbcoq [10]. F4 and GB are the fastest (up to our knowledge) available programs that compute Gröbner bases. Gbcoq is slow in general but is proved to be correct (in Coq), and we adapted it to our specific problem to be efficient. The automated proofs concern equalities and non-equalities on polynomials with coefficients and indeterminates in R or Z, and are done by reducing to Gröbner computation, via Hilbert's Nullstellensatz. We adapted also the results of [7], to allow to prove some theorems about modular arithmetics. The connection between Coq and the programs that compute Gröbner bases is done using the "external" tactic of Coq that allows to call arbitrary programs accepting xml inputs and outputs. We also produce certificates in order to make the proof scripts independant from the external programs.

1 Introduction

Proof assistants contain now more and more automatic procedures that generate proofs in specific domains. In the Coq system, several tactics exist, for example the `omega` tactic which proves inequalities between linear expressions with integer variables, the `fourier` tactic which does the same thing with real numbers, the `ring` and `field` tactic, which proves equalities between expressions in a ring or a field, the `sos` tactic which proves some inequalities on real polynomials. We describe here a new tactic, called `gb`, which proves (non-)equalities in rings using other (non-)equalities as hypotheses. For example $\forall xy : R, x^2 + xy = 0,\ y^2 + xy = 0 \Rightarrow x + y = 0$, or $\forall x : R, x^2 \neq 1 \Rightarrow x \neq 1$.

This tactic uses external efficient programs that compute Gröbner bases, and their result to produce a proof and a certificate.

We wrote such a tactic several years ago [9], but using only the gbcoq program, which were rather slow. So the tactic remained experimental and was not included in the Coq system. There are also similar tactics in other proof systems: in hol-light, John Harrison wrote a program that computes Gröbner bases to prove polynomial equalities, specially in arithmetics [7]. This program was adapted in Isabelle by Amine Chaieb and Makarius Wenzel for the same task [2]. We show on examples that our tactic is faster.

This paper is organized as follow. In section 2 we explain the mathematical method we use to reduce the problem to Gröbner bases computations. In section 3 we detail the tactic and the way it builds a proof in Coq. In section 4 we show how we connected Coq to the specialized programs that computes Gröbner bases. Section 5 details the complete tactics that proves also non-equalities, and section 6 shows how to produce certificates and then save time in the proof script. In section 7 we give some examples of utilisations of the tactic in algebra, geometry and arithmetics, with comparisons with hol-light[6] . Section 8 contains the conclusion and perpectives of this work.

Rudnicki P, Sutcliffe G., Konev B., Schmidt R., Schulz S. (eds.);
Proceedings of the Combined KEAPPA - IWIL Workshops, pp. 67-76
[1] downloadable at `http://www-sop.inria.fr/marelle/Loic.Pottier/gb-keappa.tgz`

2 Hilbert Nullstellensatz

Hilbert Nullstellensatz shows how to reduce proofs of equalities on polynomials to algebraic computations (see for example [3] for the notions introduced in this section).

It is easy to see that if a polynomial P in $K[X_1,\ldots,X_n]$ verifies $P^r = \sum_{i=1}^{s} Q_i P_i$, with r a positive integer, Q_i and P_i also in $K[X_1,\ldots,X_n]$, then P is zero whenever polynomials $P_1,...,P_s$ are zero.

Then we can reduce the proof of $P_1 = 0,\ldots,P_s = 0 \Rightarrow P = 0$ to find $Q_1,\ldots,Q_s$ and r such that $P^r = \sum_i Q_i P_i$.

The converse is also true when K is algebraically closed: this is the Hilbert Nullstellensatz. In this case, the method is complete.

Finding $P^r = \sum_i Q_i P_i$ can be done using Gröbner bases, as we will explain now.

Recall that an *ideal* $\mathscr{I}$ of a ring is an additive sub-group of the ring such that $ax \in \mathscr{I}$ whenever $a \in \mathscr{I}$. The ideal *generated* by a family of polynomials is the set of all linear combinations of these polynomials (with polynomial coefficients).

A *Gröbner basis* of an ideal is a set of polynomials of the ideal such that their head monomials (relative to a choosen order on monomials, e.g. lexicographic order, or degree order) generates the ideal of head monomials of all polynomials in the ideal. The main property of a Gröbner basis is that it provides a test for the membership to the ideal: a polynomial is in the ideal iff its euclidian *division* by the polynomials of the basis gives a zero remainder. The division process is a generalisation of the division of polynomials in one variable: to divide a polynomial P by a polynomial $aX^\alpha - Q$ we write $P = aX^\alpha S + T$ where T contains no monomial that is multiple of X^α. Then change P with $QS+T$ and repeat divison. The last non zero T is the remainder of the division. To divide a polynomial by a family of polynomials, we repeat this process with each polynomial of the family. In general, the remainder depends on the order we use the polynomials of the family. But with a Gröbner basis, this remainder is unique (this is a characteristic property of Gröbner basis).

2.1 Method 1: how to find $Q_1,\ldots,Q_s$ such that $1 = \sum_i Q_i P_i$

Compute a Gröbner base of the polynomials $\{tP_i - e_i,\ e_i e_j,\ e_i t\}_{i,j}$ (where $t, e_1,\ldots,e_s$ are new variables) with an order such that $t > X_i > e_i$.

Suppose that, in this basis, there is a polynomial of the form $t - \sum_i Q_i e_i$. This polynomial is then in the ideal generated by $\{tP_i - e_i,\ e_i e_j,\ e_i t\}_{i,j}$, so is a linear combination of these polynomials:

$t - \sum_i Q_i e_i = \sum_i h_i(tP_i - e_i) + \sum_{ij} g_{ij} e_i e_j + \sum_i k_i e_i t$

e_i are formal variables, so we can substitute formally e_i with tP_i, and we obtain $t(1 - \sum_i Q_i P_i) = 0 + t^2(\sum_{ij} g_{ij} P_i P_j + \sum_i k_i P_i)$.

Then the coefficient of t in this equation must be zero: $1 - \sum_i Q_i P_i = 0$, and we are done.

Note that the polynomials $\{e_i t,\ e_i e_j\}$ are not necessary, but their presence much speed up the computation of the Gröbner basis[2].

2.2 Method 2: how to find $Q_1,\ldots,Q_s$ and r such that $P^r = \sum_i Q_i P_i$

Use the standard trick: search to write $1 = \sum_i h_i P_i + h(1 - zP)$ $(*)$, where z is a new variable. This can be done with the previous method. Suppose we succeed. Let r be the max degree in z of polynomials h_i.

Substitute formally z with $1/P$, and multiply the equation (*) by P^r. Then we obtain $P^r = \sum_i Q_i P_i$, as required, where $Q_i = P^r h_i[z \leftarrow 1/P]$

[2] thanks to Bernard Mourrain for this trick

2.3 Completness

It is easy to see that methods 1 and 2 are complete in the sense that if $P^r = \sum_i Q_i P_i$ holds, there will find such an equation:

- method 1: suppose $1 - \sum_i Q_i P_i = 0$. Then $t = \sum_i Q_i t P_i$, and $t - \sum_i Q_i e_i = \sum_i Q_i (t P_i - e_i)$. Hence $t - \sum_i Q_i e_i$ belongs to the ideal of which we have computed a Gröbner basis. Because of the order we have choosen on variables, this implies that there is a polynomial $t - \sum_i h_i e_i$ in the Gröbner basis.
- method 2: suppose $P^r = \sum_i Q_i P_i$. We have $1 - z^r P^r = (1 + zP + \ldots + z^{r-1} P^{r-1})(1 - zP)$. Replacing P^r with $\sum_i Q_i P_i$ we obtain $1 = z^r (\sum_i Q_i P_i) + (1 + zP + \ldots + z^{r-1} P^{r-1})(1 - zP)$.

2.4 Example

Take $p = x + y,\ p_1 = x^2 + xy,\ p_2 = y^2 + xy$. With the previous method, the Gröbner basis is:

$$
\begin{array}{l}
t - zye_0 - zxe_0 - z^2 e_1 - z^2 e_2 - e_0 \\
y^2 e_0 - x^2 e_0 + zye_1 - zxe_2 - e_1 + e_2 \\
yxe_0 + x^2 e_0 + zye_2 + zxe_2 - e_2 \\
e_0^2 \\
xe_1 - ye_2 \\
e_0 e_1, e_1^2, e_0 e_2, e_1 e_2, e_2^2
\end{array}
$$

we obtain $r = 2,\ Q_1 = 1,\ Q_2 = 1$, and then $(x+y)^2 = 1 \times (x^2 + xy) + 1 \times (y^2 + xy)$. Which proves that $x^2 + xy = 0,\ y^2 + xy = 0 \Rightarrow x + y = 0$.

3 Proof in Coq

Coq [11] is a proof assistant based on type theory, where we can interactively build proofs of *goals*, which are logical assertions of the form $\forall H_1 : T_1, \ldots, \forall H_n : T_n, C(H1, \ldots, H_n)$. Using tactics, we can simplify the goal, while the system builds the corresponding piece of proof.

Typically we will treate goals of the form:

```
x : Z
y : Z
H : x ^ 2 + x * y = 0
H0 : y ^ 2 + x * y = 0
============================
 x + y = 0
```

Here hypotheses are variables belonging in a ring or a field, and equalities between polynomials.

We explain now how to compute and use the Nullstellensatz equation to build a proof of this goal in Coq. The steps are: syntaxification, Gröbner basis computation, and building the proof from the Nullstellensatz equation.

3.1 Syntaxification

We begin by building polynomials from the three equations in this goal. This is done in the tactic language of Coq (LTAC, which is a meta-language for computing tactics and executing them) by first computing the list of variables:

```
lv = (cons y (cons x nil))
```

and the list of polynomials:

```
lp = (cons (Add (Pow (Var 2) 2) (Mul (Var 2) (Var 1)))
     (cons (Add (Pow (Var 1) 2) (Mul (Var 2) (Var 1)))
     (cons (Sub (Add (Var 2) (Var 1)) (Const 0 1))
      nil)))
```

Variables are represented by their rank in the list of variables. Polynomials are elements of an inductive type, and we can recover the equations by interpreting them in Z with the list of variables. For example,

```
(interpret (Add (Pow (Var 2) 2) (Mul (Var 2) (Var 1)))
           lv)
```

evaluates in `x^2 + x * y`.

We used parts of the code of the `sos`[8] tactic, written by Laurent Théry.

3.2 Calling Gröbner basis computation

We call the external program gb (see section 4) with the list of polynomials; here we choose the program F4 to compute Gröbner basis:

```
external "./gb" "jcf2" lp
```

The result is the term:

```
(cons
   (Pow
     (Add
        (Add Zero
           (Mul
              (Add (Add Zero (Mul (Const 0 1) (Const 1 1)))
                 (Mul (Const 1 1) (Pow (Var 1) 1))) (Const 1 1)))
        (Mul (Const 1 1) (Pow (Var 2) 1)))
      2)
(cons (Const 1 1) (cons(Const 1 1) (cons (Const 1 1) nil))))
```

which has the structure

```
(cons (Pow p d) (cons c lq))
```

such that the Nullstellensatz equation holds:

$$cp^d = \sum_{q_i \in lq} q_i p_i$$

Here, we have $lq = q_1, q_2$, $q_1 = q_2 = 1$

3.3 Building the proof from the Nullstellensatz equation

After interpreting the polynomials q_1 and q_2 in Z using the original list of variables, we get and prove easily the goal

```
1 * (x + y)^2 = 1 * (x ^ 2 + x * y) + 1 * (y ^ 2 + x * y)
```

by the ring tactic.

To prove the original goal, it is now sufficient to rewrite `x^2 + x * y` and `y^2 + x * y` by 0, getting `1 * (x + y)^2 = 0`, and, using a simple lemma, we get `x + y = 0` and we are done.

4 Connecting F4, GB, and gbcoq to Coq

Coq allows to call arbitrary external programs via a function called "external". It sends Coq terms in xml format (i.e. as tree) to the standard output of the external program, and gets its standard output (also in xml format) as a resulting Coq term. We use this function to compute a Gröbner basis of a list of polynomials, via a single interface to three specialized programs: F4, GB, and gbcoq. This interface, called "gb" is written in ocaml. It translates the list of polynomials given as standard input in xml format in the format of the choosen program (F4, GB or gbcoq), call it with the good arguments, get its result (a Gröbner basis, if no error occured), selects its useful information, translates it in xml and sends it as result to standard output. More precisely:

- F4 is a C library, and has only an interface for Maple. We wrote a simple parser of polynomials to use it on command line, helped by J.C. Faugère.
- GB is also written in C and has a command line interface, or accept inputs in a file; with a Maple-like syntax for polynomials.
- Gbcoq is written in ocaml, so is integrated to gb. This program uses an Buchberger-like algorithm which has been extracted from Coq. So it is proven to be correct. We added recently an optimisation which reduces drastically the time to compute Nullstellensatz equations: each time we add a new polynomial during the completion via the reduction of critical pairs, we divide the polynomial that we want to test if it is in the ideal, by the current family of polynomials. If this gives zero, then we stop:, and return the Nullstellensatz coefficients, deduced from the divisions we made. More we also try its powers (up to a parametrized limit). Then, when we have computed the whole Gröbner basis, we can compute the Nullstellensatz coefficients, without having to verify that the remaining critical pairs reduce to zero. More, this is often the case that the polynomial reduces to zero with a partial Gröbner basis! The time is sometimes divided by 1000 with such a technique, and always much reduced. Note that such an improvement cannot be made in a blackbox program such as the programs of JC Faugère, which are free but not opensource.

5 The gbR and gbZ tactics in Coq

We wrote two tactics: gbR for real numbers, gbZ for integers. The set of integer is not a field, but we can simulate computations in the field of rational numbers using only integers. In this case, the Nullstellensatz equation become $cp^d = \sum_i q_i p_i$, where c is an integer, and the q_i have integer coefficients.

We can allow negations of equations in the conclusion. For example $xy = 1 \Rightarrow x \neq 0$. The trick is to replace $x \neq 0$ with $x = 0 \Rightarrow 1 = 0$, which is equivalent to add a new equation in hypotheses, and replace the equation to prove with $1 = 0$.

In the case of real numbers, we can allow also negations of equations in hypotheses. For example $x^2 \neq 1 \Rightarrow x \neq 1$. This can be done by introducing new variables, remarking that $p \neq 0 \Leftrightarrow \exists t, p * t = 1$.

In the example, this gives $t(x^2-1)=1 \Rightarrow x \neq 1$. The negation in conclusion can be removed and leads to $t(x^2-1)=1,\ x-1=0 \Rightarrow 1=0$, which is proven using the Nullstellensatz equation $1=1\times(t(x^2-1)-1)+(t+tx)\times(x-1)$

Finally, the tactics use first the program F4. If it fails (for memory limits), then the tactics try GB. If it fails too, then the tactics uses gbcoq. We have also specialised tactics, allowing the user to choose which program to use, between F4, GB, and gbcoq. Indeed, experiments show that no one is better than others.

6 Certificates

Once the Nullstellensatz equation is computed, we can change the proof script, replacing the tactic gb with a similar tactic, called "check_gb" which will not call external programs, but instead it will take as arguments all the components of the Nullstellensatz equation. So, next time we will execute the proof script, for compilation for example, it will not need external Gröbner computation[3]. Let us give an example. Suppose we want to prove:

```
Goal forall x y z:R, x^2+x*y=0 -> y^2+x*y=0 -> x+y=0.
```

we execute the tactic gbR, which proves the goal, and prints these lines in the standard output of Coq:

```
(* with JC.Faugere algorithm F4 *)
gbR_begin; check_gbR
(x + y - 0)
(List.cons (x * (x * 1) + x * y) (List.cons (y * (y * 1) + x * y) List.nil))
(List.cons y (List.cons x List.nil))

(lceq
   (Pow
      (Add
         (Add Zero
            (Mul
               (Add (Add Zero (Mul (Const 0 1) (Const 1 1)))
                  (Mul (Const 1 1) (Pow (Var 1) 1))) (Const 1 1)))
         (Mul (Const 1 1) (Pow (Var 2) 1))) 2)
   (lceq (Const 1 1) (lceq (Const 1 1) (lceq (Const 1 1) lnil))))
 .
```

Then, we can replace the line calling gbR with these tactics lines, which contains no more than the components of the needed Nullstellensatz equation $(x+y)^2=1\times(x^2+xy)+1\times(y^2+xy)$, and then need much less time to evaluate, because it doesn't need Gröbner basis computation.

7 Examples

In this section we give several examples of use of the tactics gbR and gbR.

[3] thanks to Julien Narboux for this suggestion

7.1 Algebra

The following examples uses the symetric expressions of coefficients with roots of a polynomial.

First in degree 3: if x,y,z are the three complex roots of $X^3 + a*X^2 + b*X + c$ then we have $a = -(x+y+z)$, $b = x*y+y*z+z*x$, and $c = -x*y*z$. And then we can prove that $x+y+z = 0 \Rightarrow x*y+y*z+z*x = 0 \Rightarrow x*y*z = 0 \Rightarrow x = 0$, because then the polynomial becomes X^3, and has only 0 as a root.

```
Require gbZ.

Goal forall x y z:Z,
  x+y+z=0 -> x*y+y*z+z*x=0 -> x*y*z=0 -> x=0.
gbZ.
Qed.
```

More complicated, the same thing in degrees 4 and 5:

```
Goal forall x y z u:Z,
  x+y+z+u=0 ->
  x*y+y*z+z*u+u*x+x*z+u*y=0 ->
  x*y*z+y*z*u+z*u*x+u*x*y=0 ->
  x*y*z*u=0 -> x=0.
gbZ.
Qed.

Goal forall x y z u v:Z,
  x+y+z+u+v=0 ->
  x*y+x*z+x*u+x*v+y*z+y*u+y*v+z*u+z*v+u*v=0->
  x*y*z+x*y*u+x*y*v+x*z*u+x*z*v+x*u*v+y*z*u+y*z*v+y*u*v+z*u*v=0->
  x*y*z*u+y*z*u*v+z*u*v*x+u*v*x*y+v*x*y*z=0 ->
  x*y*z*u*v=0 -> x^5=0.
gbZ.
Qed.
```

Last example takes less than 1s with F4 and GB, and gbcoq. With hol-light, it takes 1s.

7.2 Geometry

Desargues theorem is too complicated to be proved with Gröbner bases. But Pappus theorem can. We formalize in Coq the set of points in the real plane:

```
Open Scope R_scope.
Record point:Type:={
 X:R;
 Y:R}.
```

Then we give two definitions of colinearity of three points (the theorem is false if we use only the second definition, because of degenerated configurations):

```
Definition colinear(C A B:point):=
  exists a:R,
 (X C)=a*(X A)+(1-a)*(X B) /\ (Y C)=a*(Y A)+(1-a)*(Y B).

Definition colinear2(A B C:point):=
  (X A)*(Y B)+(X B)*(Y C)+(X C)*(Y A)
  =(Y B)*(X C)+(Y C)*(X A)+(Y A)*(X B).
```

Then we state and prove the Pappus theorem, in a specialized (but without lost of generality) configuration:

```
Lemma pappus: forall A B C A' B' C' D E F:point,
  (X A')=0 -> (X B')=0-> (X C')=0 ->
  (Y A)=0 -> (Y B)=0 -> (Y C) = 0 ->
  colinear D A B' -> colinear D A' B ->
  colinear E A C' -> colinear E A' C ->
  colinear F B C' -> colinear F B' C ->
  colinear2 D E F.
...
gbR_choice 2.
Qed.
```

In this example, F4 fails, GB takes 9s, and gbcoq takes 3s. We also tried hol-light with this example, which takes 77s:

```
./hol

prioritize_int();;

let t1 = Unix.time();;

int_ideal_cofactors
[`XD -( x4 * XA )`;
`YD -((&1 - x4) * YB1)`;
`XD -( (&1 - x3) * XB)`;
`YD - (x3 * YA1)`;
` XE - x2 * XA`;
`YE -  (&1 - x2) * YC1`;
` XE - (&1 - x1) * XC`;
` YE - x1 * YA1`;
` XF - x0 * XB`;
` YF -  (&1 - x0) * YC1`;
` XF - (&1 - x) * XC`;
` YF - x * YB1`]
` XD * YE + XE * YF + XF * YD -(YE * XF + YF * XD + YD * XE)`;;
```

```
Unix.time()-.t1;;
```

The general case of Pappus theorem is too complicated to compute.

7.3 Arithmetics

Following the idea of [7], we can prove statements about coprimality, gcd and divisions. We have to do some work for that, because the tactic gbZ is not sufficient. But the problem is again an ideal membership one, then solvable by Gröbner basis computation. We have written a tactic doing that, called `gbarith`. Here are examples of its use in Coq:

```
Definition divides(a b:Z):= exists c:Z, b=c*a.
Definition modulo(a b p:Z):= exists k:Z, a - b = k*p.
Definition ideal(x a b:Z):= exists u:Z, exists v:Z, x = u*a+v*b.
Definition gcd(g a b:Z):= divides g a /\ divides g b /\ ideal g a b.
Definition coprime(a b:Z):= exists u:Z, exists v:Z, 1 = u*a+v*b.

Goal forall a b c:Z, divides a (b*c) -> coprime a b -> divides a c.
gbarith.
Qed.

Goal forall m n r:Z, divides m r -> divides n r -> coprime m n -> divides (m*n) r.
gbarith.
Qed.

Goal forall x y a n:Z, modulo (x^2) a n -> modulo (y^2) a n -> divides n ((x+y)*(x-y)).
gbarith.
Qed.
```

7.4 Computation times, comparison with hol-light

Previous examples, and more we made, show that no one among F4, GB, gbcoq and is better than others. hol-light is sometimes better than F4 and GB, but gbcoq is much better than hol-light. The reason is simple: we often stop computations before obtaining a Gröbner basis.

8 Conclusion

The ”external” tactic of Coq is a very good tool to use efficient programs to produce proofs in specific domains. We have shown how to use efficient Gröbner bases computations in this context. The use of certificates should be developped to reduce time of re-verification of proofs. The certificate can be written explicitely in the proof script, as we have shown here, but it could be stored in a cache. We have shown the interest of using external programs, but also their limits, as soon as it is impossible or difficult to adapt them to specific use of proof systems. We plan to investigate other decisions procedures, for example polynomial system solving, to produce new tactics in the same spirit.

Acknowledgements: we thank anonymous referees for their suggestions on the redaction of this paper and bibliographical completions.

References

[1] Bruno Buchberger. Bruno buchberger's phd thesis 1965: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. Journal of Symbolic ComputationVolume 41, Issues 3-4, Logic, Mathematics and Computer Science: Interactions in honor of Bruno Buchberger (60th birthday),, 2006.

[2] Amine Chaieb and Makarius Wenzel. Context aware calculation and deduction. In Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger, editors, *Calculemus/MKM*, volume 4573 of *Lecture Notes in Computer Science*, pages 27–39. Springer, 2007.

[3] David Eisenbud. Commutative algebra with a view toward algebraic geometry. Graduate Texts in Mathematics 150. Springer-Verlag, 1999.

[4] Jean-Charles Faugère. Gb. `http://fgbrs.lip6.fr/jcf/Software/Gb/index.html`.

[5] Jean-Charles Faugère. A new efficient algorithm for computing grobner bases (f4). volume 139, issues 1-3 of *Journal of Pure and Applied Algebra*, pages 61–88, 1999.

[6] John Harrison. Towards self-verification of hol light. In Ulrich Furbach and Natarajan Shankar, editors, *Proceedings of the third International Joint Conference, IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191, Seattle, WA, 2006. Springer-Verlag.

[7] John Harrison. Automating elementary number-theoretic proofs using gröbner bases. In Frank Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction, CADE 21*, volume 4603 of *Lecture Notes in Computer Science*, pages 51–66, Bremen, Germany, 2007. Springer-Verlag.

[8] John Harrison. Verifying nonlinear real formulas via sums of squares. In Klaus Schneider and Jens Brandt, editors, *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 102–118, Kaiserslautern, Germany, 2007. Springer-Verlag.

[9] Loïc Pottier Jérôme Créci. Gb: une procédure de décision pour le système coq. Journées Francaises des Langages Applicatifs, Sainte-Marie-de-Ré, pages http://jfla.inria.fr/2004/actes/actes–jfla–2004.tar.gz, 2004.

[10] Loïc Pottier Laurent Théry. gbcoq, 1998. `http://www-sop.inria.fr/croap/CFC/Gbcoq.html`.

[11] The Coq Development Team. The coq proof assistant, 2008. `http://coq.inria.fr/V8.1pl3/refman/index.html`.

Transforming and Analyzing Proofs in the CERES-system *

Stefan Hetzl Alexander Leitsch Daniel Weller
Bruno Woltzenlogel Paleo
Institute of Computer Languages (E185), Vienna University of Technology

Abstract

Cut-elimination is the most prominent form of proof transformation in logic. The elimination of cuts in formal proofs corresponds to the removal of intermediate statements (lemmas) in mathematical proofs. Cut-elimination can be applied to *mine* real mathematical proofs, i.e. for extracting explicit and algorithmic information. The system CERES (cut-elimination by resolution) is based on automated deduction and was successfully applied to the analysis of nontrivial mathematical proofs. In this paper we focus on the input-output environment of CERES, and show how users can interact with the system and extract new mathematical knowledge.

1 Introduction

Cut-elimination introduced by Gentzen [6] is the most prominent form of proof transformation in logic and plays an important role in automating the analysis of mathematical proofs. The removal of cuts corresponds to the elimination of intermediate statements (lemmas) from proofs resulting in a proof which is analytic in the sense, that all statements in the proof are subformulas of the result. Therefore, the proof of a *combinatorial statement* is converted into a purely *combinatorial proof*.

In a formal sense Girard's famous analysis of van der Waerden's theorem [7] consists in the application of cut-elimination to the proof of Fürstenberg and Weiss (which uses topological arguments) with the "perspective" of obtaining van der Waerden's elementary proof. Indeed, an application of a complex proof transformation like cut-elimination by humans requires a goal oriented strategy.

CERES [4, 5] is a cut-elimination method that is based on resolution. The method roughly works as follows: The structure of an **LK**-proof containing cuts is mapped to an unsatisfiable set of clauses $\mathscr{C}$ (the *characteristic clause set*). A resolution refutation of $\mathscr{C}$, which is obtained using a first-order theorem prover, serves as a skeleton for the new proof which contains only atomic cuts (AC normal form). In a final step also these atomic cuts can be eliminated, provided the (atomic) axioms are valid sequents; but this step is of minor mathematical interest and of low complexity. In the system CERES[1] this method of cut-elimination has been implemented. The extension of CERES from **LK** to **LKDe**, a calculus containing definition introductions and equality rules (see [10] and [1]), moved the system closer to real mathematical proofs. The system CERES has been applied successfully to a well known mathematical proof, namely to Fürstenberg's proof of the infinity of primes [2]; it was shown that the elimination of topological arguments from the proof resulted in Euclid's famous proof. Though CERES did not (yet) produce substantial mathematical proofs previously unknown, the analysis of Fürstenberg's proof demonstrates the potential of the method to handle nontrivial mathematics.

The main task of the CERES-method is *proof transformation*, not just proof verification. For the latter one there are powerful higher-order systems like Isabelle[2] (see [12]) and Coq[3] (see [11]); these

Rudnicki P, Sutcliffe G., Konev B., Schmidt R., Schulz S. (eds.);
Proceedings of the Combined KEAPPA - IWIL Workshops, pp. 77-91

*Supported by the Austrian Science Fund (project P19875)

[1] available at http://www.logic.at/ceres/

[2] available at http://www.cl.cam.ac.uk/research/hvg/Isabelle/

[3] available at http://coq.inria.fr/

systems have been used successfully to verify complicated and famous mathematical proofs like this of the four color theorem (see http://research.microsoft.com/~gonthier/). The core algorithm of CERES, however, works on **LK**-proofs and performs cut-elimination. For this reason we have developed the higher proof language HLK which is close to the sequent calculus **LK**, making translations to **LK** easy and efficient.

In this paper we present the input-output environment of CERES and illustrate the interaction of users with the system. In the first step proofs are formalized in HLK. The formalized proof is then compiled to **LKDe** and analyzed by the CERES-algorithm, and finally the resulting atomic cut normal form can be viewed by Prooftool. Moreover, from the normal form a Herbrand sequent can be extracted which contains mathematical information in a more condensed form. We illustrate all phases of proof specification and analysis by an example, the tape proof of Christian Urban (see also [1]).

2 System overview

Figure 1 sketches how HLK, CERES and ProofTool can be used by a mathematician to analyze existing mathematical proofs and obtain new ones. According to the labels in the edges of Figure 1, the following

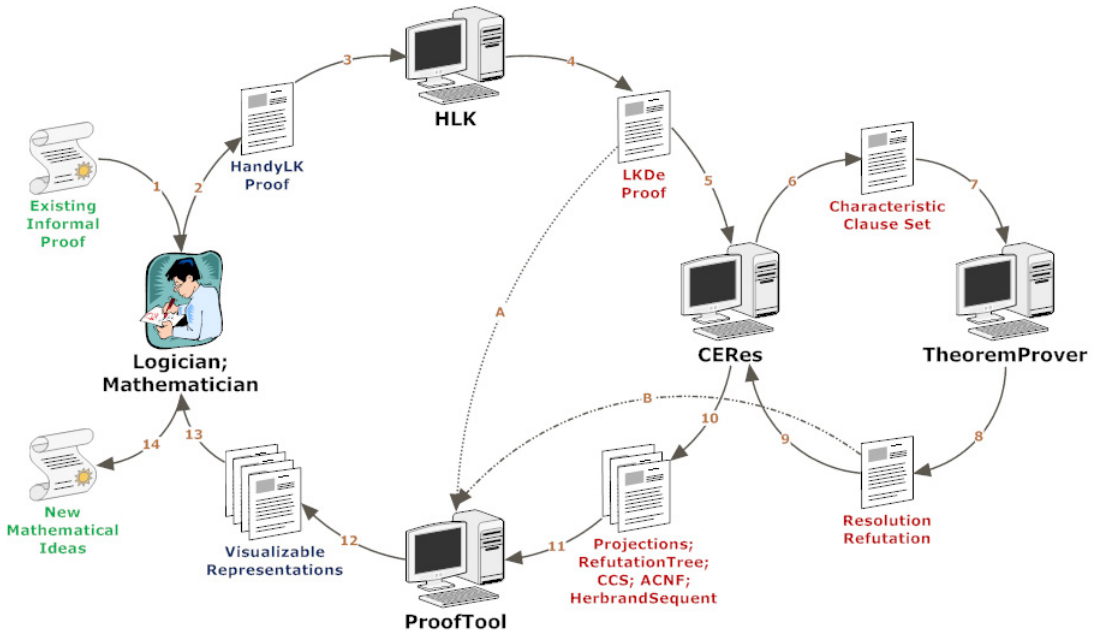


Figure 1: Working with HLK, CERES and ProofTool.

steps are executed within the system and in interaction with the user:

1. The user (intended to be a mathematician with a background in logic) selects an interesting informal mathematical proof to be transformed and analyzed. Informal mathematical proofs are proofs in natural language, as they usually occur in mathematics.

2. The user writes the selected proof in *HandyLK*, a higher proof language, intermediary between natural mathematical language and formal calculi.

3. The *HandyLK* proof is input to the compiler HLK.

4. HLK generates a formal proof in sequent calculus **LKDe**.

5. The formal proof is input to CERES, which is responsible for all sorts of proof transformations, including cut-elimination.
6. CERES extracts from the formal proof a *characteristic clause set*, which contains clauses formed from ancestors of cut-formulas in the formal proof.
7. The characteristic clause set is then input to a *resolution theorem prover*, e.g. Otter[4] or Prover9[5].
8. The resolution theorem prover outputs a refutation of the characteristic clause set.
9. CERES receives the refutation, which will be used as a skeleton for the transformed proof in atomic-cut normal form (ACNF).
10. CERES outputs the grounded refutation in a tree format and the characteristic clause set. Moreover it extracts projections from the formal proof and plugs them into the refutation in order to generate the ACNF. The projections and the ACNF are also output. A Herbrand sequent is obtained from the instantiation information of the quantifiers of the end-sequent (resulting in an (equationally) valid sequent) and output as well. The Herbrand sequent typically summarizes the creative content of the ACNF. For details, see [8].
11. All outputs and inputs of CERES can be opened with ProofTool.
12. ProofTool, a graphical user interface, renders all proofs, refutations, projections, sequents and clause sets so that they can be visualized by the user.
13. The information displayed via ProofTool is analyzed by the user.
14. Based on his analysis, the user can formulate new mathematical ideas, e.g. a new informal direct proof corresponding to the ACNF.

3 Proof analysis with CERES

Our calculus **LKDe** is based on standard **LK** with permutation, contraction and weakening, atomic axioms and multiplicative rules. For example, the following rules deal with the $\wedge$-connective:

$$\frac{\Gamma \vdash \Delta, A \quad \Pi \vdash \Lambda, B}{\Gamma, \Pi \vdash \Delta, \Lambda, A \wedge B} \wedge : r \qquad \frac{A, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta} \wedge : l1 \qquad \frac{A, \Gamma \vdash \Delta}{B \wedge A, \Gamma \vdash \Delta} \wedge : l2$$

In addition, to bring the calculus closer to mathematical practice, definition introduction and equality handling rules are used:

Let A be a first-order formula with the free variables $x_1, \ldots, x_k$ (denoted by $A(x_1, \ldots, x_k)$) and P be a *new* k-ary predicate symbol (corresponding to the formula A). Then the *definition introduction* rules are:

$$\frac{A(t_1, \ldots, t_k), \Gamma \vdash \Delta}{P(t_1, \ldots, t_k), \Gamma \vdash \Delta} \text{def}_P : l \qquad \frac{\Gamma \vdash \Delta, A(t_1, \ldots, t_k)}{\Gamma \vdash \Delta, P(t_1, \ldots, t_k)} \text{def}_P : r$$

for arbitrary sequences of terms $t_1, \ldots, t_k$. Definition introduction is a simple and very powerful tool in mathematical practice. Note that the introduction of important concepts and notations like groups, integrals etc. can be formally described by introduction of new symbols. There are also definition introduction rules for new function symbols which are of similar type.

[4] available at http://www-unix.mcs.anl.gov/AR/otter/
[5] available at http://www.cs.unm.edu/~mccune/prover9/

The *equality rules* (also called *paramodulation rules*) are:

$$\frac{\Gamma_1 \vdash \Delta_1, s=t \quad A[s]_\Lambda, \Gamma_2 \vdash \Delta_2}{A[t]_\Lambda, \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2} =: l1 \quad \frac{\Gamma_1 \vdash \Delta_1, t=s \quad A[s]_\Lambda, \Gamma_2 \vdash \Delta_2}{A[t]_\Lambda, \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2} =: l2$$

for inference on the left and

$$\frac{\Gamma_1 \vdash \Delta_1, s=t \quad \Gamma_2 \vdash \Delta_2, A[s]_\Lambda}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2, A[t]_\Lambda} =: r1 \quad \frac{\Gamma_1 \vdash \Delta_1, t=s \quad \Gamma_2 \vdash \Delta_2, A[s]_\Lambda}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2, A[t]_\Lambda} =: r2$$

on the right, where Λ denotes a set of positions of subterms where replacement of s by t has to be performed. We call $s=t$ the *active equation* of the rules.

We will now introduce the CERES system by presenting the analysis of a proof from [13]; it was formalized in **LKDe** and analyzed by CERES in [1]. The end-sequent formalizes the statement: on a tape with infinitely many cells which are all labelled by 0 or by 1 there are two cells labelled by the same number. $f(x) = 0$ expresses that the cell nr. x is labelled by 0. Indexing of cells is done by number terms defined over $0, 1$ and $+$. The proof φ below uses two lemmas: (1) there are infinitely many cells labelled by 0 and (2) there are infinitely many cells labelled by 1. These lemmas are eliminated by CERES and a more direct argument is obtained in the resulting proof φ'.

3.1 Specifying proofs in *HandyLK*

In this section, we introduce the higher proof language *HandyLK* by presenting its most important features and syntax by means of our example proof. *HandyLK* is based on the idea that many aspects of writing **LKDe** proofs can be automatized. It supports a many-sorted first-order language.

Before starting to write proofs, it is first necessary to define the language and the function and predicate definitions one intends to use. In our example, we deal with a language with one sort (natural numbers) and the constants mentioned above. In *HandyLK*, we write:

```
define type nat;
define constant 0, 1 of type nat;
define infix function + of type nat,nat to nat with weight 100;
define function f of type nat to nat;
```

In the definition of infix functions like $+$, we may set a weight that allows terms to be written in the usual mathematical way, where superfluous brackets may be dropped. In addition to the constants, we define some variables that will be used in the proof. Here, the function f is the labelling function assigning labels to tape cells.

```
define variable k, l, n, p, q, x, n_0, n_1 of type nat;
```

We also define the axioms we will use. Note that `:-` represents $\vdash$ in *HandyLK*.

```
define axiom :- k + l = l + k;
define axiom :- k + (l + n) = (k + l) + n;
define axiom k = k + (1 + l) :- ;
define axiom :- k = k;
```

Finally, we introduce some predicate definitions.

```
define predicate A by all x ( f(x) = 0 or f(x) = 1 );
define predicate I by all n ex k f( n + k ) = x;
```

Note that in the definition of I, x is a free variable. The free variables determine the arity of the defined predicate, and can be instantiated by parameters. If P is a defined predicate, F is its defining formula and $\bar{x}$ are the free variables of F, then the predicate definition is interpreted as $\forall \bar{x}(P(\bar{x}) \iff F)$. In our example, the predicate A is the assumption: All cells are labelled either 0 or 1. The predicate $I(x)$ states that there are infinitely many cells labelled x.

We now turn to formalizing our example proof φ. In **LKDe**, φ is

$$\dfrac{\dfrac{\begin{matrix}(\tau)\\ A \vdash I(0), I(1)\end{matrix} \quad \begin{matrix}(\varepsilon(0))\\ I(0) \vdash \exists p \exists q (p \neq q \wedge f(p) = f(q))\end{matrix}}{A \vdash \exists p \exists q (p \neq q \wedge f(p) = f(q)), I(1)}\ cut \quad \begin{matrix}(\varepsilon(1))\\ I(1) \vdash \exists p \exists q (p \neq q \wedge f(p) = f(q))\end{matrix}}{A \vdash \exists p \exists q (p \neq q \wedge f(p) = f(q))}\ cut$$

Here τ is a proof of the fact that either infinitely many cells are labelled 0, or infinitely many cells are labelled 1. The two applications of cut encode the application of two lemmas $\varepsilon(n)$ for $n \in \{0,1\}$: if there are infinitely many cells labelled n, then there are two cells labelled by the same number.

In *HandyLK*, we start the specification of the proof by giving an identifier, here we choose the name *the-proof*:

```
define proof the-proof
```

Next, the end-sequent is fixed:

```
  proves A :- ex p ex q ( not p = q and f(p) = f(q) );
```

Now, we begin by specifying the inferences from the bottom up. In *HandyLK*, only the active formulas of a rule have to be specified, and no structural rules (except cut) have to be written down explicitly. The HLK compiler keeps track of the formulas used in the proof, and automatically inserts the structural rules necessary to be able to apply the rules specified by the user. The first inference in φ is a cut, which is a binary rule. Due to the linear nature of the *HandyLK* language, one of the subproofs above the binary rule has to be referenced. The following proof reference clauses are available:

```
by proof ⟨proof⟩
auto propositional ⟨sequent⟩
explicit axiom ⟨sequent⟩
```

The first clause gives the name of the subproof with which to continue, the second clause states that the subproof ends in a propositional tautology whose proof should be computed automatically, and the last clause asserts that the subproof consists solely of an axiom. In our example we reference the proof $\varepsilon(1)$.

```
  with cut I(1)
    right by proof \epsilon(1);
```

This specifies an application of cut with cut formula $I(1)$, where the right subproof will be $\varepsilon(1)$. In **LKDe**, this will be a rule application

$$\dfrac{\Gamma \vdash \Delta, I(1) \quad \begin{matrix}(\varepsilon(1))\\ I(1), \Pi \vdash \Lambda\end{matrix}}{\Gamma, \Pi \vdash \Delta, \Lambda}\ cut$$

where $\Gamma, \Pi, \Delta, \Lambda$ will be populated by the appropriate formulas automatically by the HLK compiler.

We continue the *HandyLK* specification of this proof part with the left subproof of the first application of cut by encoding the second application: here, we reference both the left subproof τ and the right subproof $\varepsilon(0)$.

```
  with cut I(0)
    left by proof \tau
    right by proof \epsilon(0);
;
```

This completes the *HandyLK* specification of φ — but to be able to actually compile the proof into an **LKDe**-proof, the subproofs τ, $\varepsilon(0)$ and $\varepsilon(1)$ have to be encoded in *HandyLK*. We continue by showing the *HandyLK* rule applications in τ, which are definition introduction and quantifier rules. The proof τ starts by expanding the two occurrences of the defined predicate I:

$$\dfrac{\dfrac{A \vdash \forall n \exists k f(n+k) = 0, \forall n \exists k f(n+k) = 1}{A \vdash I(0), \forall n \exists k f(n+k) = 1}\ \mathrm{def}_I : r}{A \vdash I(0), I(1)}\ \mathrm{def}_I : r$$

In *HandyLK*, it suffices to specify the predicate to be expanded and give the auxiliary formulas of the rules, the correct main formula is then found by HLK through matching:

```
  with undef I
    :- all n ex k f( n + k ) = 0, all n ex k f( n + k ) = 1;
```

Next, the two strong quantifiers are eliminated by $\forall : r$ rules, introducing the eigenvariables n_0 and n_1:

$$\dfrac{\dfrac{A \vdash \exists k f(n_0+k) = 0, \exists k f(n_1+k) = 1}{A \vdash \exists k f(n_0+k) = 0, \forall n \exists k f(n+k) = 1}\ \forall : r}{A \vdash \forall n \exists k f(n+k) = 0, \forall n \exists k f(n+k) = 1}\ \forall : r$$

The corresponding *HandyLK* code is

```
  with all right
    :- ex k f( n_0 + k ) = 0, ex k f( n_1 + k ) = 1;
```

Again, it suffices to specify the auxiliary formulas, the correct main formulas are found by matching. The same holds for the weak quantifier rules. In our example, we instantiate them by n_1 and n_0, respectively:

$$\dfrac{\dfrac{A \vdash f(n_0+n_1) = 0, f(n_1+n_0) = 1}{A \vdash \exists k f(n_0+k) = 0, f(n_1+n_0) = 1}\ \exists : r}{A \vdash \exists k f(n_0+k) = 0, \exists k f(n_1+k) = 1}\ \exists : r$$

```
  with ex right
    :- f( n_0 + n_1 ) = 0, f( n_1 + n_0 ) = 1;
```

Of course, HLK checks whether the quantifier rule applications are legal. Our proof τ is nearly done — it remains to decompose our assumption A and to use the commutativity of $+$. We only show the latter, which is encoded with an application of paramodulation:

$$\dfrac{\vdash n_0+n_1 = n_1+n_0 \quad f(n_1+n_0) = 1 \vdash f(n_1+n_0) = 1}{f(n_0+n_1) = 1 \vdash f(n_1+n_0) = 1}\ =: l_2$$

Such a paramodulation application is common when formalizing proofs from mathematics: an equality axiom from the background theory is used as the active equation. For this reason, it receives special treatment in *HandyLK*: the active equation is simply specified in a by clause.

```
  with paramod by n_0 + n_1 = n_1 + n_0
    right f( n_1 + n_0 ) = 1 :- ;
```

This rule application can also be encoded in the general syntax for binary rules, using a proof reference:

```
  with paramod
    f( n_1 + n_0 ) = 1 :-
    left explicit axiom :- n_0 + n_1 = n_1 + n_0 ;
```

Recall that the definition of φ in *HandyLK* contained references to proofs $\varepsilon(0)$ and $\varepsilon(1)$, which show that under the assumption that there are infinitely many cells labelled 0 and 1, respectively, it follows that two cells are labelled by the same number. Clearly, these proofs have a similar structure — this fact can be exploited in *HandyLK* by writing a meta proof that can be instantiated with some specific terms:

```
define proof \epsilon
  with meta term i of type nat;

  proves
    I(i) :- ex p ex q ( not p = q and f(p) = f(q) );
```

We omit the details of the specification of the meta proof $\varepsilon(i)$. Additionally, *HandyLK* supports the definition of proofs in a recursive way, which is a convenient way to encode sequences of proofs. This feature was used to encode a sequence of proofs for the formula scheme from [3].

3.2 XML format for proofs

The programs in the CERES system do not work directly with proofs written in *HandyLK* — these proofs are compiled into a flexible XML format using the HLK compiler. XML is a well known data representation language which allows the use of arbitrary and well known utilities for editing, transformation and presentation and standardized programming libraries. We are interested in tree-style proofs, formulas and terms. Data is structured in trees in XML, therefore the encoding and decoding of these objects is very straightforward.

The format is flexible in the sense that it only assumes that proofs are trees or directed acyclic graphs of rule applications that are labelled by sequents. In our practice, it has been used successfully to encode **LKDe** and resolution proofs. The following abbreviated XML code represents the proof φ:

```
<proof symbol="the-proof" calculus="LK">
 <rule symbol="c:r" type="contrr" param="2">
  <sequent>...</sequent>
  <rule symbol="cut" type="cut">
   <sequent>...</sequent>
   <rule symbol="\pi:r" type="permr" param="(1 2)">
    <sequent>...</sequent>
    <rule symbol="cut" type="cut">
     <sequent>...</sequent>
     <prooflink symbol="\tau"/>
     <prooflink symbol="\epsilon(0)"/>
    </rule>
   </rule>
   <prooflink symbol="\epsilon(1)"/>
  </rule>
 </rule>
</proof>
```

The *symbol* attributes are used for the visual presentation of the proof trees: they allow the user to identify proofs and rules. The *param* attributes contain additional rule information, in the example they specify which formulas are to be contracted (by the contraction rule `c:r`) and how the formulas are to be permuted (by the permutation rule `\pi:r`). The `prooflink` tags reference other proofs by their name and allow the representation of proofs as DAGs.

In the above example, the formulas contained in the sequents have been left out. To give an example of how a formula is encoded in the XML format, consider the following XML code corresponding to the formula $R(c, f(c)) \wedge P(c)$:

```
<conjunctiveformula type="and">
 <constantatomformula symbol="R">
  <constant symbol="c"/>
  <function symbol="f">
   <constant symbol="c"/>
  </function>
 </constantatomformula>
 <constantatomformula symbol="P">
  <constant symbol="c"/>
 </constantatomformula>
</conjunctiveformula>
```

Data is organized in our XML format in a proofdatabase containing a number of proofs. The proofdatabase may also contain

1. a list of defined predicates and their definitions,
2. a list of sequents specifying the axioms of the background theory in which the proofs are written,
3. arbitrary lists of sequents identified by some symbol (e.g. to store a Herbrand sequent).

3.3 The CERES method

To prepare for the following section, we will now introduce the theoretical background for the CERES method of cut-elimination, which is used by the CERES system to compute a proof in atomic cut normal form and, in the end, a Herbrand sequent.

The central idea of CERES consists in extracting a so-called characteristic clause set from a proof, and then using a resolution refutation of this set to obtain a proof with only atomic cuts. We consider the proofs in **LKDe** as directed trees with nodes which are labelled by sequents, where the root is labelled by the end-sequent. According to the inference rules, we distinguish binary and unary nodes. In an inference

$$\frac{\nu_1 : S_1 \quad \nu_2 : S_2}{\nu : S}\ x$$

where ν is labelled by S, ν_1 by S_1 and ν_2 by S_2, we call ν_1, ν_2 *predecessors* of ν. Similarly ν' is predecessor of ν in a unary rule if ν' labels the premiss and ν the consequent. Then the *predecessor relation* is defined as the reflexive and transitive closure of the relation above. Every node is predecessor of the root, and the axioms have only themselves as predecessors. For a formal definition of the concepts we refer to [4] and [5]. A similar relation holds between *formula occurrences* in sequents. Instead of a formal definition we give an example.

Consider the rule:

$$\frac{\forall x.P(x) \vdash P(a) \quad \forall x.P(x) \vdash P(b)}{\forall x.P(x) \vdash P(a) \wedge P(b)}\ \wedge : r$$

The occurrences of $P(a)$ and $P(b)$ in the premiss are *ancestors* of the occurrence of $P(a) \wedge P(b)$ in the consequent. $P(a)$ and $P(b)$ are called *auxiliary formulas* of the inference, and $P(a) \wedge P(b)$ the *main formula*. $\forall x.P(x)$ in the premisses are ancestors of $\forall x.P(x)$ in the consequent. Again the *ancestor relation* is defined by reflexive transitive closure.

Let Ω be the set of all occurrences of cut-formulas in sequents of an **LKDe**-proof φ. The cut-formulas are not ancestors of the formulas in the end-sequent, but they might have ancestors in the axioms (if the cuts are not generated by weakening only). The construction of the characteristic clause set is based on the ancestors of the cuts in the axioms. Note that *clauses* are just defined as atomic sequents. We define a set of clauses $\mathscr{C}_\nu$ for every node ν in φ inductively:

- If ν is an occurrence of an axiom sequent $S(\nu)$, and S' is the subsequent of $S(\nu)$ containing only the ancestors of Ω then $\mathscr{C}_\nu = \{S'\}$.
- Let ν' be the predecessor of ν in a unary inference then $\mathscr{C}_\nu = \mathscr{C}_{\nu'}$.
- Let ν_1, ν_2 be the predecessors of ν in a binary inference. We distinguish two cases
  (a) The auxiliary formulas of ν_1, ν_2 are ancestors of Ω. Then
  $$\mathscr{C}_\nu = \mathscr{C}_{\nu_1} \cup \mathscr{C}_{\nu_2}.$$
  (b) The auxiliary formulas of ν_1, ν_2 are not ancestors of Ω. Then
  $$\mathscr{C}_\nu = \mathscr{C}_{\nu_1} \times \mathscr{C}_{\nu_2}.$$
  where $\mathscr{C} \times \mathscr{D} = \{C \circ D \mid C \in \mathscr{C}, D \in \mathscr{D}\}$ and $C \circ D$ is the merge of the clauses C and D.

The *characteristic clause set* $\mathrm{CL}(\varphi)$ of φ is defined as $\mathscr{C}_{\nu_0}$, where ν_0 is the root.

Theorem 1. *Let φ be a proof in* **LKDe**. *Then the clause set* $\mathrm{CL}(\varphi)$ *is equationally unsatisfiable.*

Remark. A clause set $\mathscr{C}$ is equationally unsatisfiable if $\mathscr{C}$ does not have a model where $=$ is interpreted as equality over a domain.

Proof. This proof first appeared in [1]. Let ν be a node in φ and $S'(\nu)$ the subsequent of $S(\nu)$ which consists of the ancestors of Ω (i.e. of a cut). It is shown by induction that $S'(\nu)$ is **LKDe**-derivable from $\mathscr{C}_\nu$. If ν_0 is the root then, clearly, $S'(\nu_0) = \vdash$ and the empty sequent $\vdash$ is **LKDe**-derivable from the axiom set $\mathscr{C}_{\nu_0}$, which is just $\mathrm{CL}(\varphi)$. As all inferences in **LKDe** are sound over equational interpretations (where new symbols introduced by definition introduction have to be interpreted according to the defining equivalence), $\mathrm{CL}(\varphi)$ is equationally unsatisfiable. Note that, without the rules $=\!:\, l$ and $=\!:\, r$, the set $\mathrm{CL}(\varphi)$ is just unsatisfiable. Clearly the rules $=\!:\, l$ and $=\!:\, r$ are sound only over equational interpretations. □

The next steps in CERES are

(1) the computation of the proof projections $\varphi[C]$ w.r.t. clauses $C \in \mathrm{CL}(\varphi)$,

(2) the refutation of the set $\mathrm{CL}(\varphi)$, resulting in an RP-tree γ, i.e. in a deduction tree defined by the inferences of resolution and paramodulation, and

(3) "inserting" the projections $\varphi[C]$ into the leaves of γ.

For step (1) we skip in φ all inferences where the auxiliary resp. main formulas are ancestors of a cut. Instead of the end-sequent S we get $S \circ C$ for a $C \in \mathrm{CL}(\varphi)$.

Step (2) consists in ordinary theorem proving by resolution and paramodulation (which is equationally complete). For refuting $\mathrm{CL}(\varphi)$ any first-order resolution prover can be used. By the completeness of the methods we find a refutation tree γ as $\mathrm{CL}(\varphi)$ is unsatisfiable by Theorem 1.

Step (3) makes use of the fact that, after computation of the simultaneous most general unifier of the inferences in γ, the resulting tree γ' is actually a derivation in **LKDe**. Indeed, after computation of the simultaneous unifier, paramodulation becomes $=: l$ and $=: r$ and resolution becomes cut in **LKDe**. Note that the definition rules, like the logical rules, do not appear in γ'. Now for every leaf ν in γ', which is labelled by a clause C' (an instance of a clause $C \in \mathrm{CL}(\varphi)$) we insert the proof projection $\varphi[C']$. The result is a proof with only atomic cuts.

The proof projection is only sound if the proof φ is skolemized, i.e. there are no strong quantifiers (i.e. quantifiers with eigenvariable conditions) in the end-sequent. If φ is not skolemized a priori it can be transformed into a skolemized proof φ' in polynomial (at most quadratic) time; for details see [3].

For illustration, consider the following example:

$\varphi =$

$$\dfrac{\begin{array}{cc}\varphi_1 & \varphi_2\end{array}}{(\forall x)(P(x) \to Q(x)) \vdash (\exists y)(P(a) \to Q(y))}\ cut$$

$\varphi_1 =$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P(u)^\star \vdash P(u) \quad Q(u) \vdash Q(u)^\star}{P(u)^\star, P(u) \to Q(u) \vdash Q(u)^\star} \to : l}{P(u) \to Q(u) \vdash (P(u) \to Q(u))^\star} \to : r}{P(u) \to Q(u) \vdash (\exists y)(P(u) \to Q(y))^\star} \exists : r}{(\forall x)(P(x) \to Q(x)) \vdash (\exists y)(P(u) \to Q(y))^\star} \forall : l}{(\forall x)(P(x) \to Q(x)) \vdash (\forall x)(\exists y)(P(x) \to Q(y))^\star} \forall : r$$

$\varphi_2 =$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P(a) \vdash P(a)^\star \quad Q(v)^\star \vdash Q(v)}{P(a), (P(a) \to Q(v))^\star \vdash Q(v)} \to : l}{(P(a) \to Q(v))^\star \vdash P(a) \to Q(v)} \to : r}{(P(a) \to Q(v))^\star \vdash (\exists y)(P(a) \to Q(y))} \exists : r}{(\exists y)(P(a) \to Q(y))^\star \vdash (\exists y)(P(a) \to Q(y))} \exists : l}{(\forall x)(\exists y)(P(x) \to Q(y))^\star \vdash (\exists y)(P(a) \to Q(y))} \forall : l$$

We have $\mathrm{CL}(\varphi) = \{P(u) \vdash Q(u);\ \vdash P(a);\ Q(v) \vdash\}$. The resolution refutation δ of $\mathrm{CL}(\varphi)$

$$\dfrac{\dfrac{\vdash P(a) \quad P(u) \vdash Q(u)}{\vdash Q(a)}\ R \qquad Q(v) \vdash}{\vdash}\ R$$

does the job of refuting $\mathrm{CL}(\varphi)$. By applying the most general unifier σ of δ, we obtain a ground refutation $\gamma = \delta\sigma$:

$$\dfrac{\dfrac{\vdash P(a) \quad P(a) \vdash Q(a)}{\vdash Q(a)}\ R \qquad Q(a) \vdash}{\vdash}\ R$$

This will serve as a skeleton for a proof in ACNF. To complete the construction, we combine the skeleton with the following projections (grounded by σ):

$\varphi(C_1) =$

$$\dfrac{\dfrac{\dfrac{P(a)\vdash P(a)\quad Q(a)\vdash Q(a)}{P(a),P(a)\to Q(a)\vdash Q(a)}\to:l}{P(a),(\forall x)(P(x)\to Q(x))\vdash Q(a)}\forall:l}{P(a),(\forall x)(P(x)\to Q(x))\vdash(\exists y)(P(a)\to Q(y)),Q(a)}w:r$$

$\varphi(C_2) =$

$$\dfrac{\dfrac{\dfrac{\dfrac{P(a)\vdash P(a)}{P(a)\vdash P(a),Q(v)}w:r}{\vdash P(a)\to Q(v),P(a)}\to:r}{\vdash(\exists y)(P(a)\to Q(y)),P(a)}\exists:r}{(\forall x)(P(x)\to Q(x))\vdash(\exists y)(P(a)\to Q(y)),P(a)}w:l$$

$\varphi(C_3) =$

$$\dfrac{\dfrac{\dfrac{\dfrac{Q(a)\vdash Q(a)}{P(a),Q(a)\vdash Q(a)}w:l}{Q(a)\vdash P(a)\to Q(a)}\to:r}{Q(a)\vdash(\exists y)(P(a)\to Q(y))}\exists:r}{Q(a),(\forall x)(P(x)\to Q(x))\vdash(\exists y)(P(a)\to Q(y))}w:l$$

The composition of skeleton and projections yields

$\varphi(\gamma) =$

$$\dfrac{\dfrac{\dfrac{\varphi(C_2)\qquad\varphi(C_1)}{B\vdash C,P(a)\quad P(a),B\vdash C,Q(a)}cut\qquad\varphi(C_3)}{B,B\vdash C,C,Q(a)\qquad Q(a),B\vdash C}cut}{B,B,B\vdash C,C,C}\text{ contractions}}{B\vdash C}$$

where $B=(\forall x)(P(x)\to Q(x))$, $C=(\exists y)(P(a)\to Q(y))$. Clearly, $\varphi(\gamma)$ is a proof of the end-sequent of φ in ACNF.

3.4 Cut-elimination, proof visualization and Herbrand sequent extraction with CERES

After compiling the *HandyLK* source to XML with HLK, it is possible to perform proof analysis using the CERES tool. In our example, we are interested in extracting a Herbrand sequent from a cut-free proof of the end-sequent of φ. To this end, the CERES method is used, which extracts an unsatisfiable set of clauses from the proof and refutes it. In theory, any resolution prover can be used to refute this set of clauses; Table 1 lists the format conversions currently supported by our implementation.

Table 1: Format conversions supported by CERES

From	To
XML	TPTP
XML	Otter input
XML	Prover9 input
Otter proof object	XML
Prover9 output	XML

In our example, we use Otter to find a resolution refutation. Running CERES yields the following characteristic clause set in Otter input format

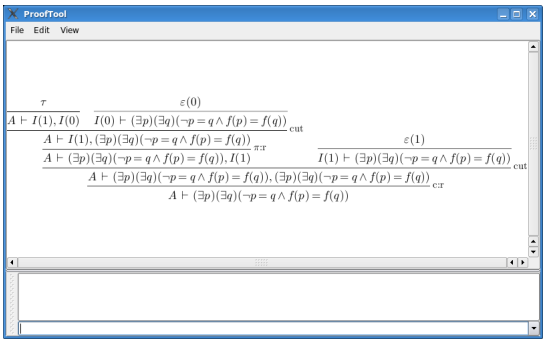
```
-> =("f"(+(x4, x5)), "0"), =("f"(+(x5, x4)), "1"). % (C1)
=("f"(+(x4, x6)), "0"), =("f"(+(+(+(x4, x6), "1"), x7)), "0") ->. % (C2)
=("f"(+(x4, x6)), "1"), =("f"(+(+(+(x4, x6), "1"), x7)), "1") ->. % (C3)
```

which corresponds to the set of clauses

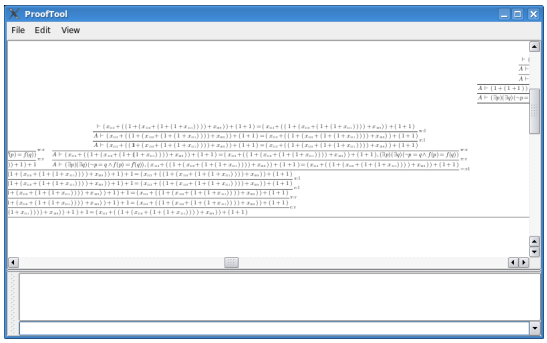
$$\begin{aligned}\mathrm{CL}(\pi) = \{ \quad & \vdash f(x_4+x_5)=0, f(x_5+x_4)=1; \\ & f(x_4+x_6)=0, f(((x_4+x_6)+1)+x_7)=0 \vdash; \\ & f(x_4+x_6)=1, f(((x_4+x_6)+1)+x_7)=1 \vdash \quad \}\end{aligned}$$

which is refutable in arithmetic.

The Otter proof object resulting from the successful refutation is then converted into a resolution tree in XML format. According to the CERES method, this refutation is then transformed into an **LKDe** proof φ' of the end-sequent of φ in ACNF. φ' consists of 1262 rules, and is clearly too large to be depicted here (a small part of it can be seen in Figure 2(b)). From φ', a Herbrand sequent is extracted and stored in XML as a list of sequents (containing only the Herbrand sequent). All proofs and sequents computed during the analysis can be visualized using `ProofTool` as seen in Figure 2.



(a) Visualizing the input proof φ.



(b) A bird's eye view on the output proof φ'.

Figure 2: `ProofTool` screenshots

To make the Herbrand sequent extracted in our example easier to read, the terms have been simplified modulo commutativity and associativity of $+$, and the following abbreviations are used:

$$\begin{aligned} &p_1 = x+z+2, && p_2 = x+z+4, && p_3 = z+1, && p_4 = x+y+z+3, \\ &p_5 = w+x+y+z+6, && p_6 = w+x+y+z+8, && p_7 = w+x+y+z+4, \end{aligned}$$

With this, the Herbrand sequent of the ACNF of the tape proof is

$$\begin{aligned} & f(p_1)=0 \vee f(p_1)=1, f(p_2)=0 \vee f(p_2)=1, f(p_3)=0 \vee f(p_3)=1, \\ & f(p_4)=0 \vee f(p_4)=1, f(p_5)=0 \vee f(p_5)=1, f(p_6)=0 \vee f(p_6)=1, \\ & f(p_7)=0 \vee f(p_7)=1 \\ & \vdash \\ & p_1 \neq p_2 \wedge f(p_1)=f(p_2), p_3 \neq p_1 \wedge f(p_3)=f(p_1), \\ & p_3 \neq p_2 \wedge f(p_3)=f(p_2), p_1 \neq p_4 \wedge f(p_1)=f(p_4), \\ & p_5 \neq p_6 \wedge f(p_5)=f(p_6), p_7 \neq p_5 \wedge f(p_7)=f(p_5), \\ & p_7 \neq p_6 \wedge f(p_7)=f(p_6), p_4 \neq p_7 \wedge f(p_4)=f(p_7). \end{aligned}$$

Clearly, the Herbrand sequent is much smaller than the ACNF from which it is extracted. As it contains all quantifier instantiation information from the proof, it is much better suited to human interpretation than the proof itself. The Herbrand sequent we extracted from φ' can be interpreted as the following proof:

Theorem 2. *On a tape with infinitely many cells where each cell is labelled* 0 *or* 1*, there are two distinct cells that are labelled the same.*

Proof. It is easy to see that the following inequalities hold:

$$p_1 \neq p_2, \quad p_3 \neq p_1, \quad p_3 \neq p_2, \quad p_1 \neq p_4,$$
$$p_5 \neq p_6, \quad p_7 \neq p_5, \quad p_7 \neq p_6, \quad p_4 \neq p_7.$$

We may therefore delete their occurrences from the right side of the Herbrand sequent and obtain the simplified sequent

$$\begin{array}{l} f(p_1) = 0 \vee f(p_1) = 1, f(p_2) = 0 \vee f(p_2) = 1, f(p_3) = 0 \vee f(p_3) = 1, \\ f(p_4) = 0 \vee f(p_4) = 1, f(p_5) = 0 \vee f(p_5) = 1, f(p_6) = 0 \vee f(p_6) = 1, \\ f(p_7) = 0 \vee f(p_7) = 1 \\ \vdash \\ f(p_1) = f(p_2), f(p_3) = f(p_1), f(p_3) = f(p_2), f(p_1) = f(p_4), \\ f(p_5) = f(p_6), f(p_7) = f(p_5), f(p_7) = f(p_6), f(p_4) = f(p_7). \end{array}$$

Now, we assume that this sequent is false and obtain a contradiction. If the sequent is false, then all formulas on the left side are true and all formulas on the right side are false, so we may assume $f(p_i) = 0 \vee f(p_i) = 1$ for $i \in \{1,\ldots,7\}$. Let $f(p_1) = a$ with $a \in \{0,1\}$ and let $\bar{a} = 1 - a$. We have assumed that the first formula on the right side is false, so $f(p_2) = \bar{a}$. From the second formula we obtain $f(p_3) = \bar{a}$, and from the third $f(p_3) = \bar{\bar{a}} = a$, which yields the desired contradiction. □

Note that the proof extracted from the Herbrand sequent uses only 3 cells even though the sequent itself is not minimal. Still, the proof yields a minimal Herbrand sequent directly:

$$\begin{array}{l} f(p_1) = 0 \vee f(p_1) = 1, f(p_2) = 0 \vee f(p_2) = 1, f(p_3) = 0 \vee f(p_3) = 1, \\ \vdash \\ p_1 \neq p_2 \wedge f(p_1) = f(p_2), p_3 \neq p_1 \wedge f(p_3) = f(p_1), p_3 \neq p_2 \wedge f(p_3) = f(p_2). \end{array}$$

Comparing the proof obtained from the Herbrand sequent with the original proof, we have gained an important piece of information: in the original proof, it is shown that either infinitely many cells are labelled 0 or 1, while in the cut-free proof, only finitely many cells are used in the proof.

4 Open problems and future work

On the theoretical side we are working on the following two major extensions of the CERES-method: The first is an extension of CERES to second-order logic, a first step in that direction has been achieved by extending it to the fragment of second-order proofs defined by containing only quantifier-free comprehension in [9]. This extension will allow the formalization of a much broader range of mathematical proofs and yield a more convenient proof formalization. The second theoretical extension enables the method to work without skolemization whose main benefit will be the ability to eliminate single cuts or even parts of a cut which is more realistic in mathematical applications than the simultaneous elimination of all cuts.

Concerning the practical aspects, it became clear in our experiments with the system that there are two bottlenecks: 1. the formalization of the input proof and 2. finding a resolution refutation of the characteristic clause set. Accordingly also these two points are of major concern for our future work.

To ease the formalization of the input proof, we plan to enhance the capabilities of `HLK`, in particular those of the `auto propositional`-mode to cover also reasoning in equational background theories

specified by term rewriting systems. We expect this feature to greatly reduce the amount of time spent on proof formalization, as large parts of formalized proofs consist of equational reasoning. In the long term, however, an interesting option would be to use one of the available proof assistants for the formalization of proofs as these are highly developed tools. The theoretical obstacle to the use of existing proof assistants lies in the fact that a translation of the formalized proof is necessary, as proof assistants typically work in set theory or higher-order logic. Moreover, this translation should not be uniform because, depending on the formalized proof and the aims of the proof analysis, one logical translation will be more useful than another. The main practical obstacle lies in transforming proof objects generated by the proof assistant (if they are provided at all) into the sequent calculus format as described in Section 3.2. Here, a lot of work will have to go into importing the standard library of basic number systems, basic operations, data structures, etc.

The search for a refutation of the characteristic clause set turned out to be a hard problem for current theorem provers (as described in [2]). To handle this problem, we need automated theorem provers with high flexibility, allowing an interactive construction of the resolution refutation. For using several provers in order to combine their respective advantages it would be very beneficial to have a standard output format for encoding resolution refutations, a goal which is partially realized by the TPTP output format. Another approach to enhance the power of the provers is to exploit the fact that the characteristic clause sets of the CERES-method form a very specific subclass of theorem proving problems. In particular, it seems promising to store in the clause set certain information about the structure of the original proof as hints on how to find a refutation and to develop resolution refinements for CERES to use this additional information.

Another important area for future improvement is the human-readable representation of the output proofs as well as other results of the analysis (e.g. Herbrand sequents); this aspect is also critical for the method to adapt better to larger proofs. As the terms generated by cut-elimination are frequently very large it would pay out to implement term simplifiers. A general approach to the simplification of Herbrand sequents lies in computing a minimal variant of it by most general unification and a (possibly external) tautology checker. For specific theories, e.g. arithmetic, one can even do better by using term simplification algorithms based on term rewriting systems. Another useful addition is to include a flexible handling of definitions in the extraction and display of Herbrand sequents which would make them an even more powerful tool for understanding the mathematical content of a formal proof. These changes will further contribute to the creation of an interface for proof analysis which is attractive to logicians and mathematicians alike.

References

[1] Matthias Baaz, Stefan Hetzl, Alexander Leitsch, Clemens Richter, and Hendrik Spohr. Proof transformation by CERES. In *Lecture Notes in Artificial Intelligence*, volume 4108, pages 82–93. Mathematical Knowledge Management, Springer Berlin, 2006.

[2] Matthias Baaz, Stefan Hetzl, Alexander Leitsch, Clemens Richter, and Hendrik Spohr. Ceres: An Analysis of Fürstenberg's Proof of the Infinity of Primes. *Theoretical Computer Science*, 403:160–175, August 2008.

[3] Matthias Baaz and Alexander Leitsch. Cut normal forms and proof complexity. *Annals of Pure and Applied Logic*, 97(1–3):127–177, 1999.

[4] Matthias Baaz and Alexander Leitsch. Cut-elimination and Redundancy-elimination by Resolution. *Journal of Symbolic Computation*, 29(2):149–176, 2000.

[5] Matthias Baaz and Alexander Leitsch. Towards a clausal analysis of cut-elimination. *Journal of Symbolic Computation*, 41:381–410, 2006.

[6] G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210,405–431, 1934–1935.

[7] Jean-Yves Girard. *Proof Theory and Logical Complexity*. Elsevier, 1987.

[8] Stefan Hetzl, Alexander Leitsch, Daniel Weller, and Bruno Woltzenlogel Paleo. Herbrand sequent extraction. In *Lecture Notes in Artificial Intelligence*, volume 5144, pages 462–477. Mathematical Knowledge Management, Springer Berlin, 2008.

[9] Stefan Hetzl, Alexander Leitsch, Daniel Weller, and Bruno Woltzenlogel Paleo. A Clausal Approach to Proof Analysis in Second-Order Logic. In *Symposium on Logical Foundations of Computer Science (LFCS 2009)*, Lecture Notes in Computer Science. Springer, 2009. to appear.

[10] Alexander Leitsch and Clemens Richter. Equational theories in ceres. unpublished (available at `http://www.logic.at/ceres/`), 2005.

[11] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[12] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — a proof assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer Berlin, 2002.

[13] C. Urban. *Classical Logic and Computation*. PhD thesis, University of Cambridge Computer Laboratory, 2000.

Interactive Verification of Concurrent Systems using Symbolic Execution

Michael Balser, Simon Bäumler, Wolfgang Reif, and Gerhard Schellhorn
University of Augsburg

Abstract

This paper presents an interactive proof method for the verification of temporal properties of concurrent systems based on symbolic execution. Symbolic execution is a well known and very intuitive strategy for the verification of sequential programs. We have carried over this approach to the interactive verification of arbitrary linear temporal logic properties of (infinite state) parallel programs. The resulting proof method is very intuitive to apply and can be automated to a large extent. It smoothly combines first order reasoning with reasoning in temporal logic. The proof method has been implemented in the interactive verification environment KIV and has been used in several case studies.

1 Introduction

Compared to sequential programs, both the design and the verification of concurrent systems is more difficult, mainly because the control flow is more complex. Particularly, for reactive systems, not only the final result of execution but the sequence of output over time is relevant for system behavior. Finding errors by means of testing strategies is limited, because, especially for interleaved systems, an exponential amount of possible executions must be considered. The execution is nondeterministic, making it difficult to reproduce errors. An alternative to testing is the use of formal methods to specify and verify concurrent systems with mathematical rigor. Automatic methods – especially model checking – have been applied successfully to discover flaws in the design and implementation of systems. Starting from systems with finite state spaces of manageable size, research in model checking aims at mastering ever more complex state spaces and to reduce infinite state systems by abstraction. In general, systems must be manually abstracted to ensure that formal analysis terminates. An alternative approach to large or infinite state spaces are interactive proof calculi. They directly address the problem of infinite state spaces. Here, the challenge is to achieve a high degree of automation. Existing interactive calculi to reason in temporal logic about concurrent systems are generally difficult to apply. The strategy of symbolic execution, on the other hand, has been successfully applied to the interactive verification of sequential programs (e.g. Dynamic Logic [9, 11]). Symbolic execution gives intuitive proofs with a high degree of automation.

Combining Dynamic Logic and temporal logic has already been investigated, e.g. Process Logic [17] and Concurrent Dynamic Logic [16] and more recently [8, 10]. These works focus on combinations of logic, while we are interested in an interactive proof method based on symbolic execution. Symbolic execution of parallel programs has been investigated in [1], however, this approach has been restricted to the verification of pre/post conditions. Other approaches are often restricted to the verification of certain types of temporal properties. Our approach presents an interactive proof method to verify arbitrary temporal properties for parallel programs with the strategy of symbolic execution and thus promises to be intuitive and highly automatic.

Our logic is based on Interval Temporal Logic (ITL) [15]. The chop operator $\varphi; \psi$ of ITL corresponds to sequential composition and programs are just a special case of temporal formulas. In addition, we have

Rudnicki P, Sutcliffe G., Konev B., Schmidt R., Schulz S. (eds.);
Proceedings of the Combined KEAPPA - IWIL Workshops, pp. 92- 102

defined an interleaving operator $\varphi \,||\, \psi$ to interleave arbitrary temporal formulas. Our logic explicitly considers arbitrary environment steps after each system transition, which is similar to Temporal Logic of Actions (TLA) [13]. This ensures that systems are compositional and proofs can be decomposed. In total, we have defined a compositional logic which includes a rich programming language with interleaved parallel processes similar to the one of STeP [7]. Important for interactive proofs, system descriptions need not be translated to flat transition systems. The calculus directly reasons about programs.

A short overview of our logic is given in Section 2. The calculus for symbolic execution is described in Section 3. The strategy has been implemented in KIV (see Section 4). Section 6 concludes. For more details on the logic and calculus, especially on induction and double primed variables to decompose proofs, we refer to [2].

2 Logic

Similar to [15], we have defined a first order interval temporal logic with static variables a, dynamic variables A, functions f, and predicates p. Let v be a static (i.e. constants - written in small letters) or dynamic variable. Then, the syntax of (a subset of) our logic is defined

$$\begin{array}{rcl} e & ::= & a \mid A \mid A' \mid A'' \mid f(e_1,\ldots,e_n) \\ \varphi & ::= & p(e_1,\ldots,e_n) \mid \neg\,\varphi \mid \varphi_1 \wedge \varphi_2 \mid \exists\, v.\, \varphi \\ & & \mid \textbf{step} \mid \varphi_1;\varphi_2 \mid \varphi^* \\ & & \mid \lceil A_1,\ldots,A_n \rceil \mid \varphi_1 \,||^{<}\, \varphi_2 \end{array}$$

Dynamic variables can be primed and double primed. It is possible to quantify both static and dynamic variables. The chop operator $\varphi_1;\varphi_2$ directly corresponds to the sequential composition of programs. The star operator φ^* is similar to a loop. We have added an operator $\lceil A_1,\ldots,A_n \rceil$ to define an explicit frame assumption. Furthermore, operator $\varphi_1 \,||^{<}\, \varphi_2$ can be used to interleave two "processes". The basic operator $||^{<}$ gives precedence to the left process, i.e., a transition of φ_1 is executed first.

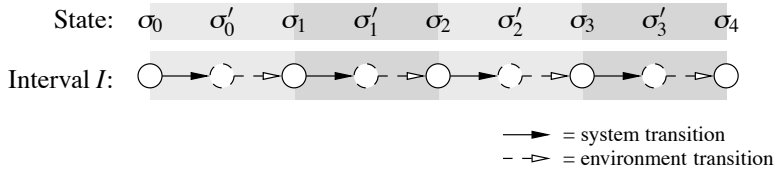


Figure 1: An interval as sequence of states

In ITL, an interval is considered to be a finite or infinite sequence of states, where a state is a mapping from variables to their values. In our setting, we introduce additional intermediate states σ_i' to distinguish between system and environment transitions. For intuition, compare the following to Figure 1. Let $\overline{n} \in \mathbb{N}^\infty$. An *interval*

$$I = (\sigma_0, \sigma_0', \sigma_1, \ldots, \sigma_{\overline{n}-1}', \sigma_{\overline{n}})$$

consists of an initial state σ_0, and a finite or infinite and possibly empty sequence of transitions $(\sigma_i', \sigma_{i+1})_{i=0}^{\overline{n}-1}$. In the intermediate states σ_i' the values of the variables after a system transition are stored. The following states σ_{i+1} reflect the states after an environment transition. In this manner, system and environment transitions alternate. An empty interval consists only of an initial state σ_0.

Given an interval I, variables are evaluated as follows:

$$\begin{array}{rcl} [\![v]\!]_I & := & \sigma_0(v) \\ [\![A']\!]_I & := & \begin{cases} \sigma_0'(A) & \text{if } |I| > 0 \\ \sigma_0(A) & \text{otherwise} \end{cases} \\ [\![A'']\!]_I & := & \begin{cases} \sigma_1(A) & \text{if } |I| > 0 \\ \sigma_0(A) & \text{otherwise} \end{cases} \end{array}$$

Static and unprimed dynamic variables are evaluated in the initial state. Primed variables are evaluated in σ_0' and double primed variables in σ_1. In the last state, i.e. if I is empty, the value of a primed or double primed variable is equal to the unprimed variable. It is assumed that after a system has terminated, the variables do not change.

The semantics of the standard ITL operators can be carried over one-to-one to our notion of an interval, and is therefore omitted here. In [15], however, assignments only restrict the assigned variables, whereas program assignments leave all other dynamic variables unchanged. This is known as a frame assumption. In our logic, we have defined an explicit frame assumption $\lceil A_1, \ldots, A_n \rceil$ which states that a system transition leaves all but a selection of dynamic variables unchanged.

$$I \models \lceil A_1, \ldots, A_n \rceil \quad \text{iff} \quad \sigma_0'(A) = \sigma_0(A) \text{ for all } A \notin \{A_1, \ldots, A_n\}$$

Details on frame assumptions can be found in [2].

The interleaving operator $\varphi \,||^< \psi$ interleaves arbitrary formulas φ and ψ. The two formulas represent sets of intervals. We have therefore defined the semantics of $\varphi||^< \psi$ relative to the interleaving $[\![I_1 \,||^< I_2]\!]$ of two concrete intervals I_1 and I_2.

$$\begin{array}{rcl} I \models \varphi \,||^< \psi & \text{iff} & \text{there exist } I_1, I_2 \\ & & \text{with } I \in [\![I_1 \,||^< I_2]\!] \text{ and } I_1 \models \varphi \text{ and } I_2 \models \psi \end{array}$$

For nonempty intervals $I_1 = (\sigma_0, \sigma_0', \sigma_1, \ldots)$, and $I_2 = (\tau_0, \tau_0', \tau_1, \ldots)$, interleaving of intervals adheres to the following recursive equations.

$$\begin{array}{rcl} [\![I_1 \,||^< I_2]\!] & = & \begin{cases} (\sigma_0, \sigma_0') \oplus [\![(\sigma_1, \ldots) \,||\, I_2]\!], & \text{if } I_1 \text{ is not blocked} \\ (\tau_0, \tau_0') \oplus [\![(\sigma_1, \ldots) \,||\, (\tau_1, \ldots)]\!], & \text{if } I_1 \text{ is blocked, } \sigma_0 = \tau_0 \\ \emptyset, & \text{otherwise} \end{cases} \\ [\![I_1 \,\big|\big|\, I_2]\!] & = & [\![I_1 \,||^< I_2]\!] \cup [\![I_2 \,||^< I_1]\!] \end{array}$$

Interval I_1 is blocked, if $\sigma_0'(blk) \neq \sigma_0(blk)$ for a special dynamic variable blk. The system transition toggles the variable to signal that the process is currently blocked (see **await** statement below). If I_1 is not blocked, then the first transition of I_1 is executed and the system continues with interleaving the remaining interval with I_2. (Function $\oplus$ prefixes all of the intervals of a given set with the two additional states.) If I_1 is blocked, then a transition of I_2 is executed instead. However, the blocked transition of I_1 is also consumed. A detailed definition of the semantics can be found in [2].

Additional common logical operators can be defined as abbreviations. A list of frequently used abbreviations is contained in Table 1, where most of the abbreviations are common in ITL. The next operator comes in two flavors. The strong next $\circ\,\varphi$ requires that there is a next step satisfying φ, the weak next $\bullet\,\varphi$ only states that if there is a next step, it must satisfy φ.

As can be seen in Table 1, the standard constructs for sequential programs can be derived. Executing an assignment requires exactly one step. The value of e is "assigned" to the primed value of A. Other variables are unchanged ($\lceil A \rceil$). Note that for conditionals and loops, the condition ψ evaluates in a single

more	$:\equiv$	**step**; true	$\Diamond\, \varphi$	$:\equiv$	**finite**; φ
last	$:\equiv$	$\neg$ **more**	$\Box\, \varphi$	$:\equiv$	$\neg \Diamond \neg\, \varphi$
inf	$:\equiv$	true; false	$\circ\, \varphi$	$:\equiv$	**step**; φ
finite	$:\equiv$	$\neg$ **inf**	$\bullet\, \varphi$	$:\equiv$	$\neg \circ \neg\ \varphi$

$A := e$	$:\equiv$	$A' = e \wedge \lceil A \rceil \wedge$ **step**
skip	$:\equiv$	$\lceil\, \rceil \wedge$ **step**
if ψ **then** φ_1 **else** φ_2	$\equiv$	$\psi \wedge (\textbf{skip}; \varphi_1) \vee \neg\, \psi \wedge (\textbf{skip}; \varphi_2)$
while ψ **do** φ	$:\equiv$	$((\psi \wedge (\textbf{skip}; \varphi))^* \wedge \Box\, (\textbf{last} \rightarrow \neg\, \psi)); \textbf{skip}$
var A **in** φ	$:\equiv$	$\Box\, A' = A \wedge \exists\, A.\ \varphi \wedge \Box\, A'' = A'$
bskip	$:\equiv$	$blk' \neq blk \wedge \lceil blk \rceil \wedge$ **step**
await φ **do** ψ	$:\equiv$	$((\neg\, \varphi \wedge \textbf{bskip})^* \wedge \Box\, (\textbf{last} \rightarrow \varphi)); \psi$
await φ	$:\equiv$	**await** φ **do skip**
$\varphi \,\|\|\, \psi$	$:\equiv$	$\varphi \,\|\|^{<}\, \psi \vee \psi \,\|\|^{<}\, \varphi$

Table 1: Frequently used temporal abbreviations

step. For local variable definitions, the local variable is quantified ($\exists\, A$), in addition, the environment cannot access the local variable ($\Box\, A'' = A'$). The global value of the variable is unchanged ($\Box\, A' = A$).

Parallel programs communicate using shared variables. In order to synchronize execution, the operator **await** ψ **do** φ can be used. A special dynamic variable *blk* is used to mark whether a parallel program is blocked. The **await** operator behaves like a loop waiting for the condition to be satisfied. While the operator waits, no variable is changed except for variable *blk* which is toggled. In other words, a process guarded with an **await** operator actively waits for the environment to satisfy its condition. Immediately after condition ψ is satisfied, construct φ is executed.

3 Calculus

Our proof method is based on a sequent calculus with calculus rules of the following form:

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \ldots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

Rules are applied bottom-up. Rule *name* refines a given conclusion $\Gamma \vdash \Delta$ with n premises $\Gamma_i \vdash \Delta_i$. We also rely heavily on rewriting with theorems of the form $\varphi \leftrightarrow \psi$. These allow to replace instances of φ with instances of ψ anywhere in a sequent.

3.1 Normal form

Our proof strategy is symbolic execution of temporal formulas, parallel programs being just a special case thereof. In Dynamic Logic, the leading assignment of a DL formula $\langle v := e; \alpha \rangle\, \varphi$ is executed as follows:

$$\frac{\dfrac{\Gamma_v^{v_0}, v = e_v^{v_0} \vdash \langle \alpha \rangle\, \varphi}{\Gamma \vdash \langle v := e \rangle\, \langle \alpha \rangle\, \varphi}\ \mathit{asg\ r}}{\Gamma \vdash \langle v := e; \alpha \rangle\, \varphi}\ \mathit{normalize}$$

The sequential composition is normalized and is replaced with a succession of diamond operators. Afterwards, the assignment is "executed" to compute the strongest postcondition $\Gamma_v^{v_0}, v = e_v^{v_0}$ (The substitution $\Gamma_v^{v_0}$ means that variable v is replaced with variable v_0 in Γ). In our setting, we normalize all the temporal

$$\frac{\varphi, \Gamma \vdash \Delta \quad \psi, \Gamma \vdash \Delta}{\varphi \vee \psi, \Gamma \vdash \Delta}\ \mathit{dis\ l} \qquad \frac{\varphi_a^{a_0}, \Gamma \vdash \Delta}{\exists\, a.\ \varphi, \Gamma \vdash \Delta}\ \mathit{ex\ l}\ (a_0 \notin \mathrm{free}(\varphi) \setminus \{a\} \cup \mathrm{free}(\Gamma, \Delta))$$

$$\frac{\tau_{A,A',A''}^{a,a,a} \vdash}{\tau, \mathbf{last} \vdash}\ \mathit{lst}\ (a \notin \mathrm{free}(\tau)) \qquad \frac{\tau_{A,A',A''}^{a_1,a_2,A}, \varphi}{\tau, \circ\, \varphi \vdash}\ \mathit{stp}\ (a_1, a_2 \notin \mathrm{free}(\tau, \varphi))$$

Table 2: Rules for executing an overall step

formulas of a given sequent by rewriting the formulas to a so called normal form which separates the possible first transitions and the corresponding temporal formulas describing the system in the next state. Afterwards, an overall step for the whole sequent is executed (see below). More formally, a program (or temporal formula) is rewritten to a formula of the following type

$$\tau \wedge \circ\, \varphi$$

where τ is a predicate logic formula that describes a transition as a relation between unprimed, primed and double primed variables while φ describes what happens in the rest of the interval. A program may also terminate, i.e., under certain conditions, the current state may be the last. Furthermore, the next transition can be nondeterministic, i.e., different τ_i with corresponding φ_i may exist describing the possible transitions and corresponding next steps. Finally, there may exist a link between the transition τ_i and system φ_i which cannot be expressed as a relation between unprimed, primed, and double primed variables in the transition alone. This link is captured in existentially quantified static variables a_i which occur in both τ_i and φ_i. The general pattern to separate the first transitions of a given temporal formula is

$$\tau_0 \wedge \mathbf{last} \vee \bigvee_{i=1}^{n} (\exists\, a_i.\ \tau_i \wedge \circ\, \varphi_i)\ .$$

We will refer to this general pattern as normal form.

3.2 Executing an overall step

Assume that the antecedent of a sequent has been rewritten to normal form. Further assume – to keep it simple – that the succedent is empty. (This can be assumed as formulas in the succedent are equivalent to negated formulas in the antecedent. Furthermore, several formulas in the antecedent can be combined to a single normal form.)

$$\tau_0 \wedge \mathbf{last} \vee \bigvee_{i=1}^{n} (\exists\, a_i.\ \tau_i \wedge \circ\, \varphi_i) \vdash$$

With the two rules *dis l* and *ex l* of Table 2, disjunction and quantification can be eliminated. For the remaining premises,

$$\tau_0 \wedge \mathbf{last} \vdash \qquad \tau_i \wedge \circ\, \varphi_i \vdash$$

the two rules *lst* and *stp* can be applied. If execution terminates, all free dynamic variables A – no matter, if they are unprimed, primed or double primed – are replaced by fresh static variables a. The result is a formula in pure predicate logic with static variables only, which can be proven with standard first order reasoning. Rule *stp* advances a step in the trace. The values of the dynamic variables A and A' in the old state are stored in fresh static variables a_1 and a_2. Double primed variables are unprimed variables in the next state. Finally, the leading next operators are discarded. The proof method now continues with the execution of φ_i.

$$\begin{array}{lccl} \textit{alw:} & \Box\,\varphi & \leftrightarrow & \varphi \wedge \bullet\,\Box\,\varphi \\ \textit{ev:} & \Diamond\,\varphi & \leftrightarrow & \varphi \vee \circ\,\Diamond\,\varphi \end{array}$$

Table 3: Rules for executing temporal logic operators $\Box$ and $\Diamond$

$$\frac{\vdash \varphi_1 \rightarrow \varphi_2}{\vdash \varphi_1; \psi \rightarrow \varphi_2; \psi}\ \textit{chp lem}$$

$$\begin{array}{lccl} \textit{chp dis:} & (\varphi_1 \vee \varphi_2); \psi & \leftrightarrow & \varphi_1; \psi \vee \varphi_2; \psi \\ \textit{chp ex:} & (\exists\, a.\ \varphi); \psi & \leftrightarrow & \exists\, a_0.\ \varphi_a^{a_0}; \psi \\ \textit{chp lst:} & (\tau \wedge \textbf{last}); \psi & \leftrightarrow & \tau^{A,A}_{A',A''} \wedge \psi \\ \textit{chp stp:} & (\tau \wedge \circ\,\varphi); \psi & \leftrightarrow & \tau \wedge \circ\,(\varphi; \psi) \end{array}$$

Table 4: Rules for executing sequential composition

3.3 Executing temporal logic

The idea of symbolic execution can be applied to formulas of temporal logic. For example, operator $\Box\,\varphi$ is similar to a loop in a programming language: formula φ is executed in every step. An appropriate rewrite rule is *alw* of Table 3. Formula φ must hold now, and in the next step $\Box\,\varphi$ holds again. To arrive at a formula in normal form, the first conjunct of the resulting formula $\varphi \wedge \bullet\,\Box\,\varphi$ must be further rewritten. The rewrite rule above corresponds to the recursive definition of $\Box\,\varphi$. Other temporal operators can be executed similarly.

3.4 Executing sequential composition

The execution of sequential composition of programs $\varphi; \psi$ is more complicated as we cannot give a simple equivalence which rewrites a composition to normal form. The problem is that the first formula φ could take an unknown number of steps to execute. Only after φ has terminated, we continue with executing ψ.

Rules for the execution of $\varphi; \psi$ are given in Table 4. In order to execute composition, the idea is to first rewrite formula φ to normal form

$$\left(\tau_0 \wedge \textbf{last} \vee \bigvee\,(\exists\, a_i.\ \tau_i \wedge \circ\,\varphi_i)\right); \psi$$

The first sub-formula φ can be rewritten with rule *chp lem* of Table 4. If it is valid that φ_1 implies φ_2, then $\varphi_1; \psi$ also implies $\varphi_2; \psi$. (This rule is a so-called congruence rule.) After rewriting the first sub-formula to normal form, we rewrite the composition operator. According to rules *chp dis* and *chp ex*, composition distributes over disjunction and existential quantification. If we apply these rules to the formula above, we receive a number of cases

$$(\tau_0 \wedge \textbf{last}); \psi \vee \bigvee\,\exists\, a_{i,0}.\ (\tau_{i,0} \wedge \circ\,\varphi_{i,0}); \psi$$

In the first case, program φ terminates, in the other cases, the program takes a step τ_i and continues with program φ_i. Rules *chp lst* and *chp stp* can be used to further rewrite the composition. Dynamic variables A stutter in the last step, and therefore the primed and double primed variables A' and A'' of τ are replaced by the corresponding unprimed variables if the first sub-formula terminates. The two rules give

$$\tau_0{}^{A,A}_{A',A''} \wedge \psi \vee \bigvee\,\exists\, a_{i,0}.\ \tau_{i,0} \wedge \circ\,(\varphi_{i,0}; \psi)$$

$$\frac{\vdash \varphi_1 \rightarrow \varphi_2}{\vdash \varphi_1 \,||^{<}\, \psi \rightarrow \varphi_2 \,||^{<}\, \psi}\ \textit{ilvl lem}$$

$$\begin{array}{lll}
\textit{ilvl dis:} & (\varphi_1 \vee \varphi_2) \,||^{<}\, \psi & \leftrightarrow \quad \varphi_1 \,||^{<}\, \psi \vee \varphi_2 \,||^{<}\, \psi \\
\textit{ilvl ex:} & (\exists\, a.\ \varphi) \,||^{<}\, \psi & \leftrightarrow \quad \exists\, a_0.\ \varphi_a^{a_0} \,||^{<}\, \psi \\
\multicolumn{3}{l}{\quad a_0 \text{ fresh with respect to } (\text{free}(\varphi) \setminus \{a\}) \cup \text{free}(\psi)} \\
\textit{ilvl lst:} & (\tau \wedge \textbf{last}) \,||^{<}\, \psi & \leftrightarrow \quad \tau^{A,A}_{A',A''} \wedge \psi \\
\textit{ilvl stp:} & (\tau \wedge \neg\, \textbf{blocked} \wedge \circ\, \varphi) \,||^{<}\, \psi & \\
 & \leftrightarrow \exists\, a_2.\ (\tau^{a_2}_{A''} \wedge \neg\, \textbf{blocked} \wedge \circ\, ((A = a_2 \wedge \varphi) \,||\, \psi)) & \\
\textit{ilvl blk:} & (\tau \wedge \textbf{blocked} \wedge \circ\, \varphi) \,||^{<}\, \psi & \\
 & \leftrightarrow \exists\, a_{2,1}.\ (\exists\, a_1.\ \tau^{a_1,a_{2,1}}_{A',A''}) \wedge (A = a_{2,1} \wedge \varphi) \,||^{<}_{b}\, \psi &
\end{array}$$

Table 5: Rules for executing left interleaving

In the first case, φ has terminated and we still need to execute ψ to arrive with a formula in normal form. In the other cases, we have successfully separated the formula into the first transition $\tau_{,0}$ and the corresponding rest of the program $\varphi_{,0}; \psi$.

3.5 Interleaving

As with sequential composition, the interleaving of programs cannot be executed directly, but the sub-formulas need to be rewritten to normal form first. The basic operator for interleaving is the left interleaving operator $\varphi \,||^{<}\, \psi$ which gives precedence to the left process. In order to execute $||^{<}$, the first sub-formula must be rewritten to normal form before the operator itself can be rewritten.

For left interleaving, the rules of Table 5 are similar to the rules for sequential composition. Congruence rule *ilvl lem* makes it possible to rewrite the first sub-formula to normal form. Similar to chop, left interleaving also distributes over disjunction (*ilvl dis*) and existential quantifiers (*ilvl ex*). If the first formula terminates, execution continues with the second (*ilvl lst*). This is similar to rule *chp lst*. Otherwise, execution depends on the first process being blocked. If it is not blocked, rule *ilvl stp* executes the transition and continues with interleaving the remaining φ with ψ. Note that the double primed variables of τ are replaced by static variables a_2 which must be equal to the unprimed variables the next time a transition of the first process is executed. This is to establish the environment transition of the first process as a relation which also includes transitions of the second. If the first process is blocked, then rule *ilvl blk* executes the blocked transition; the process actively waits while being blocked. However, the primed variables of τ are replaced by static variables a_1: the blocked transition of the first process does not contribute to the transition of the overall interleaving. Instead, a transition of the other process is executed.

$$\frac{\vdash \varphi_1 \rightarrow \varphi_2}{\vdash \psi \,||^{<}_{b}\, \varphi_1 \rightarrow \psi \,||^{<}_{b}\, \varphi_2}\ \textit{ilvlb lem}$$

$$\begin{array}{lll}
\textit{ilvlb dis:} & \psi \,||^{<}_{b}\, (\varphi_1 \vee \varphi_2) & \leftrightarrow \quad \psi \,||^{<}_{b}\, \varphi_1 \vee \psi \,||^{<}_{b}\, \varphi_2 \\
\textit{ilvlb ex:} & \psi \,||^{<}_{b}\, (\exists\, a.\ \varphi) & \leftrightarrow \quad \exists\, a_0.\ \psi \,||^{<}_{b}\, \varphi_a^{a_0} \\
\multicolumn{3}{l}{\quad a_0 \text{ fresh with respect to } (\text{free}(\varphi) \setminus \{a\}) \cup \text{free}(\psi)} \\
\textit{ilvlb lst:} & \psi \,||^{<}_{b}\, (\tau \wedge \textbf{last}) & \leftrightarrow \quad \text{false} \\
\textit{ilvlb stp:} & \psi \,||^{<}_{b}\, (\tau \wedge \circ\, \varphi) & \leftrightarrow \quad \exists\, a_{2,2}.\ \tau^{a_{2,2}}_{A''} \wedge \circ\, (\psi \,||\, (A = a_{2,2} \wedge \varphi))
\end{array}$$

Table 6: Rules for executing blocked left interleaving

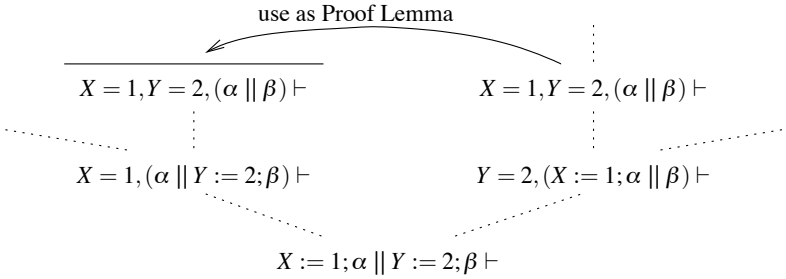


Figure 2: Example of proof lemma rule

The situation where the first process is blocked and it remains to execute a transition of the second process is represented by a derived operator $\varphi \,||_b^< \, \psi$ which is defined

$$\varphi \,||_b^< \, \psi \quad :\equiv \quad (\textbf{blocked} \wedge \circ \varphi) \,||^< \, \psi$$

and for which the rules of Table 6 are applicable. Again, the rules are very similar to the rules above. A congruence rule *ilvlb lem* ensures that the second sub-formula can be rewritten to normal form. With rules *ilvlb dis* and *ilvlb ex*, the operator distributes over disjunction and existential quantification. If the second process terminates, *ilvlb lst* is applicable, otherwise *ilvlb stp* can be applied.

Similar rules have been defined for all the operators of our logic [2]. In summary, every temporal formula can be rewritten to normal form, an overall step can be executed, and the process of symbolic execution can be repeated.

3.6 Induction and Proof Lemmas

To prove programs with loops, an inductive argument is necessary. Just as for sequential progams, we use well-founded and structural induction over datatypes. A specific rule for temporal induction over the interval is unnecessary: if a liveness property $\Diamond \varphi$ is known, the equivalence

$$\Diamond \varphi \leftrightarrow \exists N.\, N'' = N - 1 \textbf{ until } (N = 0 \wedge \varphi)$$

can be used to induce over the number of steps N it takes to reach the first state satisfying φ. To prove a safety property such a liveness property can be derived by using the equivalence $\Box \varphi \leftrightarrow \neg \Diamond \neg \varphi$. The proof is then by contradiction, assuming there is a number N of steps, after which φ is violated.

Execution of interleaved programs often leads to the same sequent occurring in different branches of the proof tree. Normally, the user would specify a lemma which can be applied to all these sequents. To simplify this, we allow that a premise of the proof tree can automatically used as lemma to close another goal. An example is shown in Figure 2. This generalisation of proof trees to (acyclic) proof graphs allows the dynamic construction of verification diagrams [18].

4 Implementation

The interactive proof method has been implemented in KIV [4], an interactive theorem prover which is available at [12]. KIV supports algebraic specifications, predicate logic, dynamic logic, and higher order logic. Especially, reasoning in predicate logic and dynamic logic is very elaborate. Support for concurrent systems and temporal logic has been added. Considerable effort has been spent to ensure that the calculus rules are automatically applied to a large extent. Almost all of the rules are invertible ensuring that, if the conclusion is provable, the resulting premises remain valid. The overall strategy is

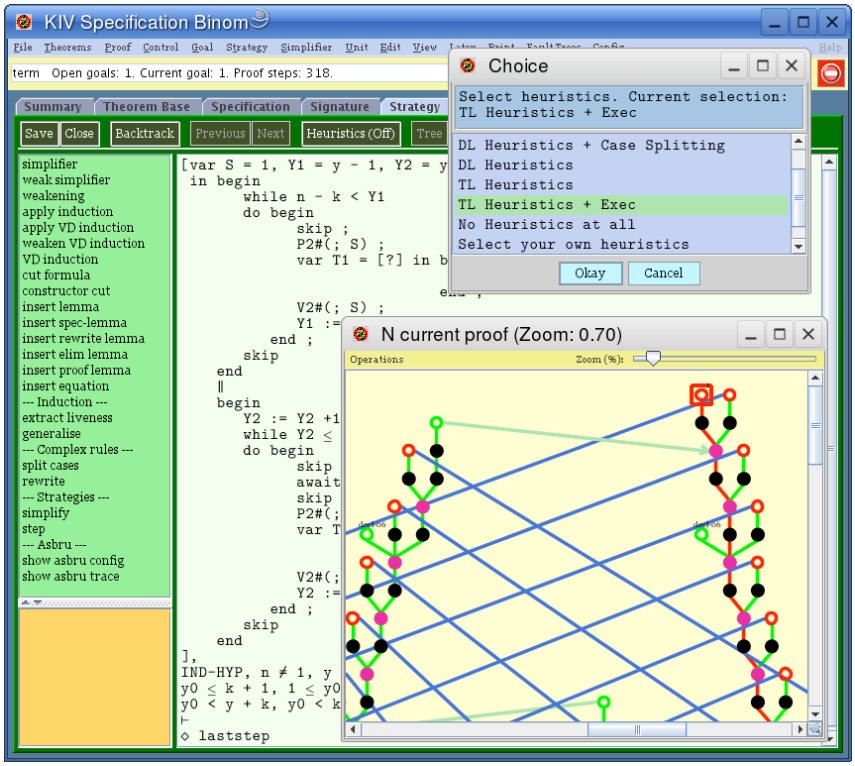


Figure 3: Verification in KIV

- to symbolically execute a given proof obligation,
- to simplify the PL formulas describing the current state,
- to combine premises with the same sequent, and
- to use induction, if a system loop has been executed.

5 Verification in KIV

Figure 3 contains a screen-shot to illustrate how to construct proofs in KIV. In the lower right, the proof graph is displayed. The main area in the large window contains the sequent under verification with the partially executed parallel program and the first order formulas describing the current state. A list of applicable proof rules is displayed to the left. Heuristics are used to automatically apply these rules. Two set of heuristics `TL Heuristics` and `TL Heuristics + Exec` implement the overall strategy of Section 4.

6 Conclusion

We have successfully carried over the strategy of symbolic execution to verify arbitrary temporal properties for concurrent systems. The proof method is based on symbolic execution, sequencing, and induction. How to symbolically execute arbitrary temporal formulas – parallel programs being just a special case thereof – has been explained in this paper.

Our proof method is easily extendable to other temporal operators. For every operator, a set of rules must be provided to rewrite the operator to normal form. With rules similar to the ones of Tables 4 and 5, we support in KIV operators for Dijkstra's choice, synchronous parallel execution, and interrupts. Furthermore, we have integrated STATEMATE state charts [3] as well as UML state charts [5] as alternative formalisms to define concurrent systems. For all of our extensions, the strategy of sequencing and induction has remained unchanged and arbitrary temporal formulas can be verified.

Using double primed variables, we have defined a compositional semantics for every operator including interleaving of formulas. This allowed us to find a suitable assumption-guarantee theorem to apply compositonal reasoning [6]. This technique is very important to verify large case studies.

The implementation in KIV has shown that symbolic execution can be automated to a large extent. We have applied the strategy to small and medium size case studies. Currently, the strategy is applied in a European project called Protocure to verify medical guidelines which can be seen as yet another form of concurrent system [14]. Overall, we believe that the strategy has the potential to make interactive proofs in (linear) temporal logic in general more intuitive and automatic.

References

[1] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.

[2] M. Balser. *Verifying Concurrent System with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction*. PhD thesis, University of Augsburg, Augsburg, Germany, 2005.

[3] M. Balser and A. Thums. Interactive verification of statecharts. In *Integration of Software Specification Techniques (INT'02)*, 2002. `http://tfs.cs.tu-berlin.de/~mgr/int02/proceedings.html`.

[4] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*. Springer LNCS 1783, 2000.

[5] S. Bäumler, M. Balser, A. Knapp, W. Reif, and A. Thums. Interactive verification of uml state machines. In *Formal Methods and Software Engineering*, number 3308 in LNCS. Springer, 2004.

[6] Simon Bäumler, Florian Nafz, Michael Balser, and Wolfgang Reif. Compositional proofs with symbolic execution. In Bernhard Beckert and Gerwin Klein, editors, *Proceedings of the 5th International Verification Workshop*, volume 372 of *Ceur Workshop Proceedings*, 2008.

[7] N. Bjørner, A. Brown, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma, and T. Uribe. Verifying temporal properties of reactive systems: A step tutorial. In *Formal Methods in System Design*, pages 227–270, 2000.

[8] D. Harel and D. Peleg. Process logic with regular formulas. *Theoretical Computer Science*, 38:307–322, 1985.

[9] David Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 2, pages 496–604. Reidel, 1984.

[10] David Harel and Eli Singerman. Computation path logic: An expressive, yet elementary, process logic. *Annals of Pure and Applied Logic*, pages 167–186, 1999.

[11] M. Heisel, W. Reif, and W. Stephan. A Dynamic Logic for Program Verification. In A. Meyer and M. Taitslin, editors, *Logical Foundations of Computer Science*, LNCS 363, pages 134–145, Berlin, 1989. Logic at Botik, Pereslavl-Zalessky, Russia, Springer.

[12] KIV homepage. `http://www.informatik.uni-augsburg.de/swt/kiv`.

[13] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[14] M.Balser, J. Schmitt, and W. Reif. Verification of medical guidelines with KIV. In *Proceedings of Workshop on AI techniques in healthcare: evidence-based guidelines and protocols (ECAI'06)*, 2006.

[15] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.

[16] D. Peleg. Concurrent dynamic logic. *J. ACM*, 34(2):450–479, 1987.

[17] R. Sherman, A. Pnueli, and D. Harel. Is the interesting part of process logic uninteresting?: a translation from pl to pdl. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 347–360, New York, NY, USA, 1982. ACM Press.

[18] Henny Sipma. *Diagram-based verification of reactive, real-time and hybrid systems*. PhD thesis, Stanford University, 1999.

Learning Techniques for Pseudo-Boolean Solving

José Santos
IST/UTL, INESC-ID, Portugal

Vasco Manquinho
IST/UTL, INESC-ID, Portugal

Abstract

The extension of conflict-based learning from Propositional Satisfiability (SAT) solvers to Pseudo-Boolean (PB) solvers comprises several different learning schemes. However, it is not commonly agreed among the research community which learning scheme should be used in PB solvers. Hence, this paper presents a contribution by providing an exhaustive comparative study between several different learning schemes in a common platform. Results for a large set of benchmarks are presented for the different learning schemes, which were implemented on `bsolo`, a state of the art PB solver.

1 Introduction

Pseudo-Boolean (PB) solvers have been the subject of recent research [4, 7, 10, 17], namely in applying techniques already successful in propositional satisfiability (SAT) solvers. In these algorithms, one of the most important techniques is the generation of no-goods during search [1, 4, 17]. This is typically done when a conflict arises, i.e, a given problem constraint is unsatisfiable. In that situation, a conflict analysis procedure is carried out and a no-good constraint is added to the constraint set so that the same conflicting assignment does not occur again.

In PB solvers, several conflict-based learning procedures have been proposed. However, there is no a consensus on which method is best for most set of instances [3]. In this paper, we start by defining the pseudo-Boolean solving problem and some operations that can be performed on PB constraints. Next, PB algorithms and conflict analysis procedures are briefly surveyed. Some implementation issues are described in section 5. Experimental results are presented for several learning procedures implemented on a state of the art pseudo-Boolean solver. Finally, conclusions are presented in section 7.

2 Preliminaries

In a propositional formula, a literal l_j denotes either a variable x_j or its complement $\bar{x}_j$. If a literal $l_j = x_j$ and x_j is assigned value 1 or $l_j = \bar{x}_j$ and x_j is assigned value 0, then the literal is said to be true. Otherwise, the literal is said to be false. A pseudo-Boolean (PB) constraint is defined as a linear inequality over a set of literals of the following normal form:

$$\sum_{j=1}^{n} a_j \cdot l_j \geq b \tag{1}$$

such that for each $j \in \{1, \cdots, n\}$, $a_j \in Z^+$ and l_j is a literal and $b \in Z^+$. It is well-known that any PB constraint (with negative coefficients, equalities or other inequalities) can be converted into the normal form in linear time [2]. In the remaining of the paper, it is assumed that all PB constraints are in normal form.

In a given constraint, if all a_j coefficients have the same value k, then it is called a cardinality constraint, since it requires that $\lceil b_i/k \rceil$ literals be true in order for the constraint to be satisfied. A

Rudnicki P, Sutcliffe G., Konev B., Schmidt R., Schulz S. (eds.);
Proceedings of the Combined KEAPPA - IWIL Workshops, pp. 103-112

PB constraint where any literal set to true is enough to satisfy the constraint, can be interpreted as a propositional clause. This occurs when the value of all a_j coefficients are greater than or equal to b.

An instance of the Pseudo-Boolean Satisfiability (PB-SAT) problem can be defined as finding an assignment to the problem variables such that all PB constraints are satisfied.

2.1 Boolean Constraint Propagation

Boolean Constraint Propagation (BCP) [6] in PB algorithms is a generalization of the application of the *unit clause rule* [6] used in propositional satisfiability (SAT) algorithms. A constraint ω is said to be *unit* if given the current partial assignment, ω is not currently satisfied and there is at least one literal l_j that if assigned value 0, ω cannot be satisfied. Hence, l_j must be assigned value 1. Let $a^u_{\max}$ denote the largest coefficient of the unassigned literals in ω. Moreover, let the *slack* s of constraint ω to be defined as follows:

$$s = (\sum_{l_j \neq 0} a_j) - b \tag{2}$$

If $a^u_{\max} > s$ then ω is a unit constraint and literal l^u_{max} must be assigned value 1. Notice that given the same partial assignment, more than one literal can be implied by the same PB constraint.

During the search, let $x_j = v_x@k$ denote the assignment of v_x to variable x_j at decision level k. In the following sections we often need to associate *dependencies* (or an *explanation*) with each implied variable assignment. Dependencies represent sufficient conditions for variable assignments to be implied. For example, let $x_j = v_x$ be a truth assignment implied by applying the unit clause rule to ω. Then the explanation for this assignment is the set of assignments associated with the false literals of ω.

2.2 Constraint Operations

It is well-known that the fundamental operation used to infer new constraints using propositional clauses is *resolution* [14]. For PB constraints, instead of resolution, the technique of cutting planes [5, 9] can be used. This operation allows the linear combination of a set of constraints, in order to generate a new constraint that is a logical consequence of the original constraints. For any two pseudo-Boolean constraints and coefficients c and c' we can combine them as follows:

$$\frac{\begin{array}{c} c(\sum_j a_j x_j \geq b) \\ c'(\sum_j a'_j x_j \geq b') \end{array}}{c\sum_j a_j x_j + c'\sum_j a'_j x_j \geq cb + c'b'} \tag{3}$$

If c or c' is non-integer, a new constraint with non-integer coefficients may result after applying the cutting plane operation. In order to obtain a new constraint with integer coefficients, the *rounding* operation can be applied as follows:

$$\frac{\sum_j a_j x_j \geq b}{\sum_j \lceil a_j \rceil x_j \geq \lceil b \rceil} \tag{4}$$

It should be noted that the rounding operation might weaken the constraint such that the number of models of the resulting constraint is larger. For example, suppose we have the constraint $\omega : 2x_1 + x_2 + x_3 \geq 3$. Applying a coefficient $c = 0.5$ to ω we get a new constraint ω':

$$\omega' : x_1 + 0.5x_2 + 0.5x_3 \geq 1.5 \tag{5}$$

Applying the rounding operation to ω' results in a new constraint $\omega'_r : x_1 + x_2 + x_3 \geq 2$. Clearly, all models of ω' are also models of ω'_r, but not all models of ω'_r are models of ω'.

Another operation that can be used on PB constraints is *reduction* [2]. This operation allows the removal of literals by subtracting the value of the coefficient from the right hand side.

Consider the constraint $\omega : x_1 + 3x_2 + 3\bar{x}_3 + 2\bar{x}_4 + x_5 \geq 7$. If reduction is applied to ω in order to remove x_1 and $\bar{x}_3$, we would get $\omega' : 3x_2 + 2\bar{x}_4 + x_5 \geq 3$. Note that if ω is a problem instance constraint, it is not possible to replace ω with ω'. Constraint ω' is a logical consequence from ω, but the converse is not true.

Algorithms to generate cardinality constraints from general pseudo-Boolean constraints can be found in [2, 4]. These algorithms find the minimum number of literals that must be set to 1 in order to satisfy the constraint. This is achieved by accumulating the literal coefficients of the constraint, in decreasing order, starting with the highest a_j. Let $m(\omega)$ denote the minimum number of literals to be set to true in order to satisfy constraint ω. Then, cardinality reduction can be defined as follows:

$$\frac{\omega : \sum_j a_j l_j \geq b}{\sum_j l_j \geq m(\omega)} \tag{6}$$

One should note that the resulting constraint is weaker than the original general pseudo-Boolean constraint. More details for cardinality reduction can be found in [4], which presents a stronger cardinality reduction.

3 Pseudo-Boolean Algorithms

Generally, pseudo-Boolean satisfiability (PB-SAT) algorithms follow the same structure as propositional satisfiability (SAT) backtrack search algorithms. The search for a satisfying assignment is organized by a decision tree (explored in depth first) where each node specifies an assignment (also known as *decision assignment*) to a previously unassigned problem variable. A *decision level* is associated with each decision assignment to denote its depth in the decision tree.

Algorithm 1 shows the basic structure of a PB-SAT algorithm. The `decide` procedure corresponds to the selection of the decision assignment thus extending the current partial assignment. If a complete assignment is reached, then a solution to the problem constraints has been found. Otherwise, the `deduce` procedure applies Boolean Constraint Propagation (and possibly other inference methods). If a conflict arises, i.e. a given constraint cannot be satisfied by extending the current partial assignment, then a conflict analysis [11] procedure is carried out to determine the level to which the search process can safely backtrack to. Moreover, a no-good constraint is also added to the set of problem constraints.

The main goal of the conflict analysis procedure is to be able to determine the correct explanation for a conflict situation and backtrack (in many cases non-chronologically) to a decision level such that the conflict does no longer occur. Moreover, a no-good constraint results from this process. However, the strategy for no-good generation that results from the conflict analysis differs between several state of the art PB-SAT solvers. These different strategies are reviewed and analysed in section 4.

Another approach to PB-SAT is to encode the problem constraints into a propositional satisfiability (SAT) problem [8] and then use a powerful SAT solver on the new formulation. Although this approach is successful for some problem instances [8], in other cases the resulting SAT formulation is much larger than the original PB formulation so that it provides a huge overhead to the SAT solver.

Algorithm 1 Generic Structure of Algorithms for PB-SAT Problem

```
while TRUE do
  if decide() then
    while deduce()=CONFLICT do
      blevel ← analyseConflict()
      if blevel ≤ 0 then
        return UNSATISFIABLE;
      else
        backtrack(blevel);
      end if
    end while
  else
    return SATISFIABLE;
  end if
end while
```

4 Conflict Analysis

Consider that the assignment of a problem variable x_j is inferred by Boolean Constraint Propagation due to a PB constraint ω_i. In this case, ω_i is referred to as the *antecedent constraint* [11] of the assignment to x_j. The *antecedent assignment* of x_j, denoted as $A^{\omega}(x_j)$, is defined as the set of assignments to problem variables corresponding to false literals in ω_i. Similarly, when ω_i becomes unsatisfied, the antecedent assignment of its corresponding conflict, $A^{\omega}(k)$, will be the set of all assignments corresponding to false literals in ω.

The implication relationships of variable assignments during the PB-SAT solving process can be expressed as an *implication graph* [11]. In the implication graph, each vertex corresponds to a variable assignment or to a conflict vertex. The predecessors of a vertex are the other vertexes corresponding to the assignments in $A^{\omega_i}(x_j)$. Next, several learning schemes are presented that result from analyzing the implication graph in a conflict situation.

4.1 Propositional Clause Learning

When a logical conflict arises, the implication sequence leading to the conflict is analysed to determine the variable assignments that are responsible for the conflict. The conjunction of these assignments represents a sufficient condition for the conflict to arise and, as such, its negation must be consistent with the PB formula [11]. This new constraint (known as the *conflict constraint*), is then added to the PB formula in order to avoid the repetition of the same conflict situation and thus pruning the search space.

When an assignment to a problem variable x_j is implied by a PB constraint ω_i, one can see it as the conjunction of $A^{\omega_i}(x_j)$ implying the assignment to x_j. Moreover, when a constraint implies a given assignment, the coefficient reduction rule can be used to eliminate all positive and unassigned literals except for the implied literal. Hence, the conflict analysis used in SAT solvers by applying a sequence of resolution steps in a backward traversal of the implication graph can be directly applied to PB formulas [11]. Additionally, techniques such as the detection of Unique Implication Points (UIPs) [11, 18] can also be directly used in PB-SAT conflict analysis. As a result, a new propositional clause is generated and added to the original formula.

In the last pseudo-Boolean solver evaluation, some PB solvers used this approach, namely `bsolo` [10] and `PBS4` (an updated version of the original `PBS` solver [1]). This strategy is simple to implement in PB solvers, since it is a straightforward generalization of the one used in SAT

solvers. Moreover, considering the use of lazy data structures [12] for clause manipulation, the overhead of adding a large number of clauses during the search is smaller than with other types of constraints.

4.2 Pseudo-Boolean Constraint Learning

The use of PB constraint learning is motivated by the fact that PB constraints are more expressive. It is known that a single PB constraint can represent a large number of propositional clauses. Therefore, the potential pruning power of PB conflict constraints is much larger than that of propositional clauses.

The operation on PB constraints which corresponds to clause resolution is the *cutting plane* operation (section 2.2). As such, to learn a general PB constraint, the conflict analysis algorithm must perform a sequence of cutting plane steps instead of a sequence of resolution steps. In each cutting plane step one implied variable is eliminated. As with the clause learning conflict analysis, a backward traversal of the implication graph is performed and the implied variables are considered in reverse order. The procedure stops when the conflict constraint is unit at a previous decision level (1UIP cut) [18].

After the conflict analysis, the algorithm uses the learned constraint to determine to which level it must backtrack as well as implying a new assignment after backtracking. Therefore, the learned constraint must be unsatisfied under the current assignment. Moreover, it must also be an assertive constraint (must become unit after backtracking).

Consider the application of a cutting plane step to two arbitrary constraints ω_1 with slack s_1 and ω_2 with slack s_2. Moreover, consider that α and β are used as the multiplying factors. In this situation, the slack of the resulting constraint, here denoted by s_r, is given by linearly combining the slacks of ω_1 and ω_2: $s_r = (\alpha \cdot s_1) + (\beta \cdot s_2)$. As such, before the application of each cutting plane step, the learning algorithm verifies if the resulting constraint is still unsatisfied under the current assignment. If it is not, the implied constraint must be reduced to lower its slack [4, 17]. This process is guaranteed to work since the repeated reduction of constraints will eventually lead to a simple clause with slack 0.

Algorithm 2 presents the pseudo-code for computing the conflict-induced PB constraint $\omega(k)$. This algorithm performs a sequence of cutting plane steps, starting from the unsatisfied constraint $\omega(c)$. Notice that this algorithm can also implement a clause learning scheme. In this case, functions `reduce1` and `reduce2` remove all non-negative literals in $\omega(k)$ and ω_i, respectively, except for the implied literal. Next, these procedures can trivially reduce the obtained constraint to a clause. In order to implement a general PB learning scheme, function `reduce2` must eliminate only enough non-negative literals in ω_i to guarantee that after the cutting plane step, the resulting constraint remains unsatisfied at the current decision level.

4.3 Other Learning Schemes

Learning general PB constraints slows down the deduction procedure because the watched literal strategy is not as efficient with general PB constraints as it is with clauses or cardinality constraints [4, 16]. Note that in a clause, as well as in a cardinality constraint, it is only necessary to watch a fixed number of literals, whereas in a general PB constraint the number of watched literals varies during the execution of the algorithm.

The approach for cardinality constraint learning used in `Galena` [4] is based on the approach described for the general pseudo-Boolean learning scheme. The difference is that a

Algorithm 2 Generic Pseudo-Boolean Learning Algorithm

```
// W corresponds to the set of constraints in the PB formula and ω(c) to the conflicting constraint
V ← {x_i | x_j corresponds to a false literal in ω(c) at current decision level};
ω(k) ← reduce1(w(c))
while TRUE do
  x_j ← removeNext(V);
  ω_i ← implyingConstraint(x_j);
  if (w_i ≠ NULL ∧ |V| > 1) then
    ω'_i ← reduce2(ω_i, ω(k));
    ω(k) ← cutResolve(ω(k), ω'_i, x_j);
    V ← V \ {x_j} ∪ {x_k | x_k corresponds to a false literal in ω'_i at current decision level}
  else
    ω(k) ← reduce3(ω(k));
    Add ω(k) to W;
    btLevel ← assertingLevel(ω(k));
    if btLevel < 0 then
      return CONFLICT;
    else
      backtrack(btLevel);
      return NO_CONFLICT;
    end if
  end if
end while
```

post-reduction procedure is carried out so that the learned constraint is reduced into a weaker cardinality constraint. In Algorithm 2 this would be done in function `reduce3`.

Finally, a hybrid learning scheme was already proposed and used in `Pueblo` [17]. The authors noted that any solver which performs PB learning can be modified to additionally perform clause learning with no significant extra overhead. Moreover, despite the greater pruning power of PB learning, clause learning has its own advantages: it always produces an assertive constraint and it does not compromise as heavily the propagation procedure as general PB learning. As such, in their solver `Pueblo`, they implement a hybrid learning method [17].

5 Implementation Issues

In implementing a pseudo-Boolean solver, several technical issues must be addressed. In this section the focus is on generating the implication graph and on the use of cutting planes for PB constraint learning.

5.1 Generating the Implication Graph

It is widely known that lazy data structures [12, 17] for constraint manipulation are essential for the solver's performance. Nevertheless, the order of propagation of variable assignments in BCP has not been thoroughly studied. In the version of `bsolo` submitted to the last PB solver evaluation (PB'07) [15], the order of propagation in the implication graph was depth-first. In this paper it is shown that by changing it to a breadth-first propagation, `bsolo` was able to solve a larger number of instances (see section 6).

When generating the implication graph in a breadth-first way, one can guarantee that there is no other possible implication graph such that the length of the longest path between the

decision assignment vertex and the conflict vertex is lower than the one considered. Therefore, the learned constraint is probably determined using a smaller number of constraints. Moreover, considering that in PB formulations the same constraint can imply more than one variable assignment, the motivation for a breadth-first propagation is larger than in SAT formulations.

5.2 Dealing with Large Coefficients

When performing general PB Learning or any learning scheme that requires performing a sequence of cutting plane steps each time a conflict occurs, the coefficients of the learned constraints may grow very fast. Note that in each cutting plane step two PB constraints are linearly combined. Given two constraints: $\sum_j a_j \cdot l_j \geq b$ and $\sum_j c_j \cdot l_j \geq d$, the size of the largest coefficient of the resulting constraint may be $\max\{b \cdot d, \max_j\{a_j\} \cdot \max_j\{c_j\}\}$ in the worst case. Therefore, it is easy to see that during a sequence of cutting plane steps the size of the coefficients of the accumulator constraint may, in the worst case, grow exponentially in the number of cutting plane steps (which is of the same order of the number of literals assigned at the current level).

One problem that may occur in the cutting plane operation is integer overflow. To avoid this problem, a maximum coefficient size was established (we used 10^6). Therefore, every time the solver performs a cutting plane step, all coefficients of the resulting constraint are checked if one of them is bigger than the established limit. If it is, the solver repeatedly divides the constraint by 2 (followed by rounding) until its largest coefficient is lower than a second maximum coefficient size (we used 10^5).

During the conflict analysis the accumulator constraint must always have negative slack. However the division rule does not preserve the slack of the resulting constraint, since it does not guarantee that the slack of the resulting constraint is equal to the slack of the original one, which can be verified in the next example where the constraint is divided by 2, followed by rounding:

$$\frac{3x_1(0@1) + 3x_2(0@1) + 3x_3(1@1) + x4(1@1) \geq 5 \quad slack = -1}{2x_1(0@1) + 2x_2(0@1) + 2x_3(1@1) + x4(1@1) \geq 3 \quad slack = 0}$$

As such, before dividing by 2 a coefficient associated with a slack contributing literal, the solver must check if it is odd. In this case it must perform a coefficient reduction step before the division (note that the coefficient reduction rule when applied to slack contributing literals preserves the slack).

$$\frac{\begin{array}{c}3x_1(0@1) + 3x_2(0@1) + 3x_3(1@1) + x4(1@1) \geq 5 \quad slack = -1 \\ \hline 3x_1(0@1) + 3x_2(0@1) + 2x_3(1@1) \geq 3 \quad slack = -1\end{array}}{2x_1(0@1) + 2x_2(0@1) + x_3(1@1) \geq 2 \quad slack = -1}$$

5.3 An Initial Classification Step

After implementing different learning schemes, it was observed that each of the versions proved to be more effective than the others for some group of instances. One can easily conclude that different learning schemes behave better (or worse) depending on some characteristics of the instances given as input. For instance, a cardinality constraint learning scheme is more appropriate to deal with an instance with a large number of cardinality constraints than a clause learning scheme.

Our goal was then to try to define an initial classification step that could choose the best fitting learning scheme to solve a given problem instance. Therefore, in a very preliminary work, algorithm C4.5 [13] was used to generate a decision tree that given a problem instance,

determines which learning scheme is more appropriate to it. The classification of each instance was done according to structural attributes of the formula, namely number of variables, literals, types of constraints, among others.

6 Experimental Results

This section presents the experimental results of applying different learning schemes to the small integer non-optimization benchmarks from the PB'07 evaluation [15]. All the learning schemes were implemented on top of `bsolo`, a state of the art PB solver. Experimental results were obtained on a Intel Xeon 5160 server (3.0Ghz, 4GB memory) running Red Hat Enterprise Linux WS 4. The CPU time limit for each instance was set to 1800 seconds.

In Table 1, results for several learning schemes are presented. Each line of the table represents a set of instances depending on their origin or encoded problem. Each column represents a version of `bsolo` for a different learning scheme. Finally, each cell denotes the number of benchmark instances that were found to be satisfiable/unsatisfiable/unknown.

The basis for our work was the `bsolo` version submitted to PB'07 solver evaluation. This version implements a clause learning scheme and is identified with `CL1`. Version `CL2` corresponds to the previous version, but with a breadth-first approach for generating the implication graph. Next, results for the general PB learning scheme `PB` and cardinality learning scheme `CARD` are presented. One should note that both the `PB` and `CARD` learning schemes are hybrid (as in `Pueblo`) and always learn an assertive clause. Preliminary results on pure PB and cardinality learning schemes were not as effective as an hybrid learning scheme. In version `COMB`, an initial classification step was used in order to select the best fitting learning scheme for each instance (see section 5.3). The training set for the `COMB` version was composed of 100 instances (out of the total of 371 instances).

It can be observed from Table 1 that the original solver was greatly improved just by changing the propagation order. Version `CL2` was able to solve more 17 instances, most of them in the tsp benchmark set. Both the `PB` and `CARD` learning schemes improve on the original version of the solver. However, in the `PB` version, the overhead associated with maintaining the additional no-good PB constraints is a drawback, in particular on the FPGA and pigeon hole instances. Overall, the `CARD` learning scheme performs much better, proving to be a nice compromise between pruning power of the generated constraints, and the underlying overhead of constraint manipulation. Finally, the `COMB` version shows that a combination of all these learning schemes allows an even better performance. Finally, one should note that improvements are essentially on unsatisfiable instances. The gain on satisfiable instances is smaller.

In Table 2, the results of the best known solvers can be checked and compared with `bsolo`. `Pueblo` is able to solve 6 more instances than the current version of `bsolo`. By using a hybrid learning scheme, `Pueblo` seems to have found a nice balance between the additional overhead of new no-good constraints (mostly clauses) and the pruning power of new no-good PB constraints. Nevertheless, the work presented in this paper shows that `bsolo` can be competitive in a large set of problem instances.

In the Pseudo-Boolean Optimization (PBO) problem, the objective is to find an assignment such that all constraints are satisfied and a linear cost function is optimized. PB solvers can be easily modified [2] in order to also tackle this problem. Table 3 presents the results of the new version of `bsolo` in comparison with other solvers. Each cell contains the following information: the number of instances for which the optimum value was found, the number of satisfiable but non-optimal solutions, the number of unsatisfiable instances and the number of

Table 1: Results for all different learning schemes on non-optimization benchmarks

Benchmark	CL1	CL2	CARD	PB	COMB
armies	5/0/7	4/0/8	6/0/6	5/0/7	7/0/5
dbst	15/0/0	15/0/0	15/0/0	15/0/0	15/0/0
FPGA	36/2/19	36/2/19	36/21/0	35/1/21	36/21/0
pigeon	0/2/18	0/2/18	0/19/1	0/1/19	0/19/1
prog. party	4/0/2	3/0/3	4/0/2	4/0/2	3/0/3
robin	3/0/3	4/0/2	2/0/4	4/0/2	4/0/2
tsp	40/33/27	40/50/10	40/44/16	40/43/17	40/49/11
uclid	1/39/10	1/40/9	1/43/6	1/43/6	1/41/8
vdw	1/0/4	1/0/4	1/0/4	1/0/4	1/0/4
wnqueen	32/68/0	32/68/0	32/68/0	32/68/0	32/68/0
Total	137/144/90	136/162/73	137/195/39	137/156/78	139/198/34

Table 2: Comparison with other solvers on non-optimization benchmarks

Benchmark	bsolo	Pueblo	minisat+	PBS4
armies	7/0/5	6/0/6	8/0/4	9/0/3
dbst	15/0/0	15/0/0	7/0/8	15/0/0
FPGA	36/21/0	36/21/0	33/3/21	26/21/10
pigeon	0/19/1	0/13/7	0/2/18	0/20/0
prog. party	3/0/3	6/0/0	5/0/1	3/0/3
robin	4/0/2	3/0/3	4/0/2	3/0/3
tsp	40/49/11	40/60/0	39/46/15	40/52/8
uclid	1/41/8	1/42/7	1/46/3	1/44/5
vdw	1/0/4	1/0/4	1/0/4	1/0/4
wnqueen	32/68/0	32/68/0	32/68/0	32/68/0
Total	139/198/34	140/203/28	130/165/76	130/205/36

unknown instances. Clearly, `bsolo` is able to prove optimality for a larger number of instances than other solvers. Additionally, due to the new learning schemes, `bsolo` is also able to find a larger number of non-optimal solutions.

7 Conclusions

Considering the disparity of results concerning the application of learning schemes in several state of the art PB solvers, the main goal of this work is to provide a contribution by implementing them in the same platform. Our results confirm that hybrid learning schemes perform better on a large set of instances. Moreover, our results show that cardinality constraint learning is more effective and robust learning scheme than others. It obtained much better results than the original clause learning scheme and also on our implementation of the PB hybrid learning scheme included in `Pueblo`. In our opinion, cardinality constraints are easier to propagate than PB constraints and are also more expressive than clauses. Therefore, this learning scheme seems a reasonable compromise between PB learning and pure clause learning.

References

[1] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Generic ILP versus specialized 0-1 ILP: An update. In *Proceedings of the International Conference on Computer Aided Design*, pages 450–457, 2002.

[2] P. Barth. *Logic-Based 0-1 Constraint Programming*. Kluwer Academic Publishers, 1995.

[3] D. Le Berre and A. Parrain. On Extending SAT-solvers for PB Problems. *14th RCRA workshop Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, July 2007.

Table 3: Comparison with other solvers on optimization benchmarks

Benchmark	bsolo	PBS4	minisat+	Pueblo
aksoy	26/52/0/1	11/63/0/5	25/45/0/9	15/64/0/0
course-ass	1/4/0/0	0/4/0/1	2/1/0/2	0/5/0/0
domset	0/15/0/0	0/15/0/0	0/15/0/0	0/15/0/0
garden	1/2/0/0	0/3/0/0	1/2/0/0	1/2/0/0
haplotype	0/8/0/0	0/8/0/0	8/0/0/0	0/8/0/0
kexu	0/40/0/0	0/40/0/0	11/29/0/0	0/40/0/0
logic-synthesis	51/23/0/0	19/55/0/0	31/41/0/2	32/42/0/0
market-split	4/16/4/16	0/20/0/20	0/20/0/20	4/16/4/16
mps-v2-20-10	12/18/1/1	7/21/1/3	10/15/1/6	10/18/1/3
numerical	12/18/0/4	12/11/0/11	10/9/0/15	14/17/0/3
primes-dimacs-cnf	69/35/8/18	69/36/8/17	79/26/8/17	75/30/8/17
radar	6/6/0/0	0/12/0/0	0/12/0/0	0/12/0/0
reduced	17/80/35/141	14/77/14/168	17/10/23/213	14/92/18/149
routing	10/0/0/0	4/6/0/0	10/0/0/0	10/0/0/0
synthesis-ptl-cmos	6/2/0/0	0/8/0/0	1/7/0/0	1/7/0/0
testset	6/0/0/0	4/2/0/0	5/1/0/0	6/0/0/0
ttp	2/6/0/0	2/6/0/0	2/6/0/0	2/6/0/0
vtxcov	0/15/0/0	0/15/0/0	0/15/0/0	0/15/0/0
wnq	0/15/0/0	0/15/0/0	0/15/0/0	0/15/0/0
Total	223/355/48/181	142/417/23/225	212/264/35/301	184/404/31/188

[4] D. Chai and A. Kuehlmann. A Fast Pseudo-Boolean Constraint Solver. In *Proceedings of the Design Automation Conference*, pages 830–835, 2003.

[5] V. Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 4:305–337, 1973.

[6] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the Association for Computing Machinery*, 5:394–397, July 1962.

[7] H. Dixon, M. Ginsberg, E. Luks, and A. Parkes. Generalizing Boolean Satisfiability I: Background and survey of existing work. *Journal of Artificial Intelligence Research*, 21:193–243, 2004.

[8] N. Eén and N. Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–25, 2006.

[9] R.E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64:275–278, 1958.

[10] V. Manquinho and J. P. Marques-Silva. Effective lower bounding techniques for pseudo-boolean optimization. In *Proceedings of the Design and Test in Europe Conference*, pages 660–665, March 2005.

[11] J. Marques-Silva and K. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer Aided Design*, pages 220–227, November 1996.

[12] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the Design Automation Conference*, pages 530–535, June 2001.

[13] J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[14] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, January 1965.

[15] O. Roussel and V. Manquinho. Third pseudo-boolean evaluation 2007. http://www.cril.univ-artois.fr/PB07, 2007.

[16] H. Sheini and K. Sakallah. Pueblo: A Modern Pseudo-Boolean SAT Solver. In *Proceedings of the Design and Test in Europe Conference*, pages 684–685, March 2005.

[17] H. Sheini and K. Sakallah. Pueblo: A Hybrid Pseudo-Boolean SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:157–181, 2006.

[18] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proceedings of the International Conference on Computer Aided Design*, pages 279–285, November 2001.

Complete Pruning Methods and a Practical Search Strategy for SOL

Hidetomo Nabeshima
University of Yamanashi, Japan

Koji Iwanuma
University of Yamanashi, Japan

Katsumi Inoue
National Institute of Informatics, Japan

Abstract

The ability to find non-trivial consequences of an axiom set is useful in many applications of AI such as theorem proving, query answering and nonmonotonic reasoning. SOL (Skip Ordered Linear) calculus is one of the most significant calculi for consequence finding, which is complete for finding the non-subsumed consequences of a clausal theory. In this paper, we propose new complete pruning methods and a practical search strategy for SOL tableaux. These methods are indispensable for practical use in many application areas. The experimental results show that these techniques improve the performance of consequence finding process greatly.

1 Introduction

The task of *consequence finding* [8] is to discover hidden theorems of a given axiom set. These theorems to be found are not given in an explicit way, but are only characterized by some desired properties. Thus, consequence finding is clearly distinguished from proof finding or theorem proving. One of the most significant calculi for consequence finding is the SOL (Skip Ordered Linear) calculus[1], which is complete for finding the non-subsumed consequences of a clausal theory. There are many application areas of SOL calculus such as *nonmonotonic reasoning*, *abduction* [1, 2], *induction* [3], *default reasoning* [4] and *query answering* [7, 6]. The original SOL-resolution calculus can be viewed as extending Loveland's model-elimination calculus [11] with a so-called *Skip* operation. Iwanuma et al. [5] subsequently reformulated SOL-resolution within the framework of connection tableaux [9, 10] and proposed several pruning methods [5, 6].

In this paper, we propose two kinds of new complete pruning methods for SOL tableau calculus, *local failure caching for length condition* and *skip-minimality*. The former is an extension of local failure caching [9, 10, 5] which is an excellent pruning technique for connection tableau calculus. The extension ensures the completeness of the calculus in case we are only interested in consequences up to a given maximal length. A user usually prefers short and simple consequences rather than long and complicated ones. Moreover, limiting the length of consequences has the effect of reducing the search space vastly. In such a situation, our proposed method accelerates the performance of the consequence finding process significantly. The latter method, Skip-minimality, restrains the use of the Skip operation and prunes many redundant SOL derivations. In addition, we describe a new practical search strategy, called *consequence iterative lengthening* (CIL), which has the effect of finding short and simple consequences preferentially in a limited time. We demonstrate the effectiveness of these methods by the experimental evaluation.

The remainder of this paper is organized as follows: Section 2 reviews the SOL tableau calculus. Section 3 and 4 introduces new pruning methods and a search strategy for consequence finding, respectively. Section 5 shows the experimental evaluation. Section 6 concludes with a summary and discussion of future work.

Rudnicki P, Sutcliffe G., Konev B., Schmidt R., Schulz S. (eds.);
Proceedings of the Combined KEAPPA - IWIL Workshops, pp. 113- 122

2 SOL Tableaux for Consequence Finding

This section reviews the SOL tableau calculus [5]. This paper assumes a first-order language (without equality). We write $\subseteq_{ms}$ to denote the inclusion relation over multisets which is defined as usual. When C and D are clauses, C *subsumes* D if there is a substitution θ such that $C\theta \subseteq_{ms} D$. We say C *properly subsumes* D if C subsumes D but D does not subsume C. For a set Σ of clauses, $\mu\Sigma$ denotes the set of clauses in Σ not properly subsumed by any clause in Σ.

Definition 1. *A production field $\mathscr{P}$ is a pair $\langle \boldsymbol{L}, Cond \rangle$, where $\boldsymbol{L}$ is a set of literals closed under instantiation, and Cond is a certain condition to be satisfied. If Cond is not specified, $\mathscr{P}$ is just denoted as $\langle \boldsymbol{L} \rangle$. A clause C belongs to $\mathscr{P} = \langle \boldsymbol{L}, Cond \rangle$ if every literal in C belongs to $\boldsymbol{L}$ and C satisfies Cond. For a set of clauses Σ, the set of logical consequences of Σ belonging to $\mathscr{P}$ is denoted as $Th_{\mathscr{P}}(\Sigma)$. A production field $\mathscr{P}$ is stable if, for any two clauses C and D such that C subsumes D, the clause D belongs to $\mathscr{P}$ only if C belongs to $\mathscr{P}$.*

Example 1. *Let $\mathscr{L} = \mathscr{L}^+ \cup \mathscr{L}^-$ be the set of all literals in the first-order language, where $\mathscr{L}^+$ and $\mathscr{L}^-$ are the positive and negative literals in the language, respectively. The following are examples of stable production fields.*

(a) $\mathscr{P}_1 = \langle \mathscr{L} \rangle$: $Th_{\mathscr{P}_1}(\Sigma)$ is the set of logical consequences of Σ.

(b) $\mathscr{P}_2 = \langle \mathscr{L}^-$, length is less than $k \rangle$: $Th_{\mathscr{P}_2}(\Sigma)$ is the set of negative clauses implied by Σ consisting of less than k literals.

In contrast, $\mathscr{P}_3 = \langle \boldsymbol{L}$, length is greater than $k \rangle$ is not a stable production field. For example, if $k = 2$ and $\boldsymbol{L} = \{p(a), q(b), \neg r(c)\}$, then $C = p(a) \vee q(b)$ subsumes $D = p(a) \vee q(b) \vee \neg r(c)$, and D belongs to $\mathscr{P}_3$ while C does not.

Note that the empty clause $\square$ is the unique clause in $\mu Th_{\mathscr{P}}(\Sigma)$ iff Σ is unsatisfiable. This means that *proof finding* is a special case of consequence finding. Stable production fields are practically important [1]. In the condition *Cond*, we can specify the maximum length of each clause, the maximum term depth of any literal in a clause, and so on.

Definition 2. *A clausal tableau T is a labeled ordered tree, where every non-root node of T is labeled with a literal. We identify a node with its label (i.e., a literal) if no confusion arises. If the immediate successors of a node are literals $L_1, \ldots, L_n$, then $L_1 \vee \cdots \vee L_n$ is called a tableau clause. The tableau clause below the root is called the start clause. T is a clausal tableau for a set Σ of clauses if every tableau clause C in T is an instance of a clause D in Σ. In this case, D is called an origin clause of C. A connection tableau is a clausal tableau such that, for every non-leaf node L except the root, there is an immediate successor labeled with $\overline{L}$. A marked tableau T is a clausal tableau such that some leaves are marked with labels* `closed` *or* `skipped`*. The unmarked leaves are called subgoals. A node N in T is said to be solved if either N itself is a marked leaf node or all leaf nodes of branches through N of T are marked. T is solved if all leaves are marked. $skip(T)$ denotes the set of literals of nodes marked with* `skipped`*.*

Notice that $skip(T)$ is a *set*, not a multiset. $skip(T)$ is also identified with a clause. We will abbreviate a marked connection tableau as a *tableau*.

Definition 3. *A tableau T is regular if no two nodes on any branch in T are labeled with the same literal. T is tautology-free if no tableau clause in T is tautology. T is complement-free if no two non-leaf nodes on any branch in T are labeled with complementary literals. A tableau T is skip-regular if there is no node L in T such that $\overline{L} \in skip(T)$. T is TCS-free (Tableau Clause Subsumption free) for a clause set Σ if no tableau clause C in T is subsumed by any clause in Σ other than the origin clauses.*

Notice that the skip-regularity applies to all over a tableau, so it is effective not only for subgoals but also for non-leaf and solved nodes.

Definition 4 (Selection Function [5]). *A selection function ϕ is a mapping assigning a subgoal to every tableau that is not solved. ϕ is said to be depth-first if ϕ selects from any tableau T a subgoal with a maximum depth. ϕ is stable under substitution if, for any tableau T and any substitution σ, $\phi(T) = \phi(T\sigma)$ holds.*

Selection functions are assumed to be stable under substitution in this paper.

Definition 5 (SOL Tableau Calculus [5]). *Let Σ be a set of clauses, C a clause, $\mathscr{P}$ a production field and ϕ a selection functions. An SOL-deduction deriving a clause S from $\Sigma + C$ and $\mathscr{P}$ via ϕ consists of a sequence of tableaux $T_0, T_1, \ldots, T_n$ satisfying that:*

(i) *T_0 consists of the start clause C only. All leaf nodes of T_0 are unmarked.*

(ii) *T_n is a solved tableau, and $skip(T_n) = S$.*

(iii) *For each T_i $(i = 0, \ldots, n)$, T_i is regular, tautology-free, complement-free, skip-regular, and TCS-free for $\Sigma \cup \{C\}$.*

(iv) *For each T_i $(i = 0, \ldots, n)$, the clause $skip(T_i)$ belongs to $\mathscr{P}$.*

(v) *T_{i+1} is constructed from T_i as follows. Select a subgoal K by ϕ, then apply one of the following rules to T_i to obtain T_{i+1}:*

  (a) **Skip:** *If $skip(T_i) \cup \{K\}$ belongs to $\mathscr{P}$, then mark K with label* `skipped`.

  (b) **Skip-factoring:** *If $skip(T_i)$ contains a literal L, and K and L are unifiable with mgu θ, then mark K with* `skipped`, *and apply θ to T_i.*

  (c) **Extension:** *Select a clause B from $\Sigma \cup \{C\}$ and obtain a variant $B' = L_1 \vee \cdots \vee L_m$ by renaming. If there is a literal L_j such that $\overline{K}$ and L_j are unifiable with mgu θ, then attach new nodes $L_1, \ldots, L_m$ to K as the immediate successors. Next, mark L_j with* `closed` *and apply θ to the extended tableau.*

  (d) **Reduction:** *If K has an ancestor node L, and $\overline{K}$ and L are unifiable with mgu θ, then mark K with* `closed`, *and apply θ to T_i.*

Theorem 1 (Soundness and Completeness of SOL [5]). *(1) If there is an SOL-deduction of a clause S from $\Sigma + C$ and $\mathscr{P}$ via ϕ, then S belongs to $Th_{\mathscr{P}}(\Sigma \cup \{C\})$.*

(2) If a clause F does not belong to $Th_{\mathscr{P}}(\Sigma)$ but belongs to $Th_{\mathscr{P}}(\Sigma \cup \{C\})$, then there is an SOL-deduction of a clause S from $\Sigma + C$ and $\mathscr{P}$ via ϕ such that S subsumes F.

Example 2. *We define a start clause C, a set Σ of clauses and a production field $\mathscr{P}$ as follows. Let ϕ be a selection function.*

$$
\begin{aligned}
C &= p(X) \vee s(X), \\
\Sigma &= \{\, q(X) \vee \neg p(X),\ \neg s(Y),\ \neg p(Z) \vee \neg q(Z) \vee r(Z) \,\}, \\
\mathscr{P} &= \langle \mathscr{L}^{+},\ \textit{length is less than } 2 \rangle .
\end{aligned}
$$

Figure 1 shows three solved tableaux that are derived by SOL-deductions from $\Sigma + C$ and $\mathscr{P}$ via ϕ. In the tableau T_a, the node $p(X)$ is skipped since the positive literal $p(X)$ belongs to $\mathscr{P}$, and $s(X)$

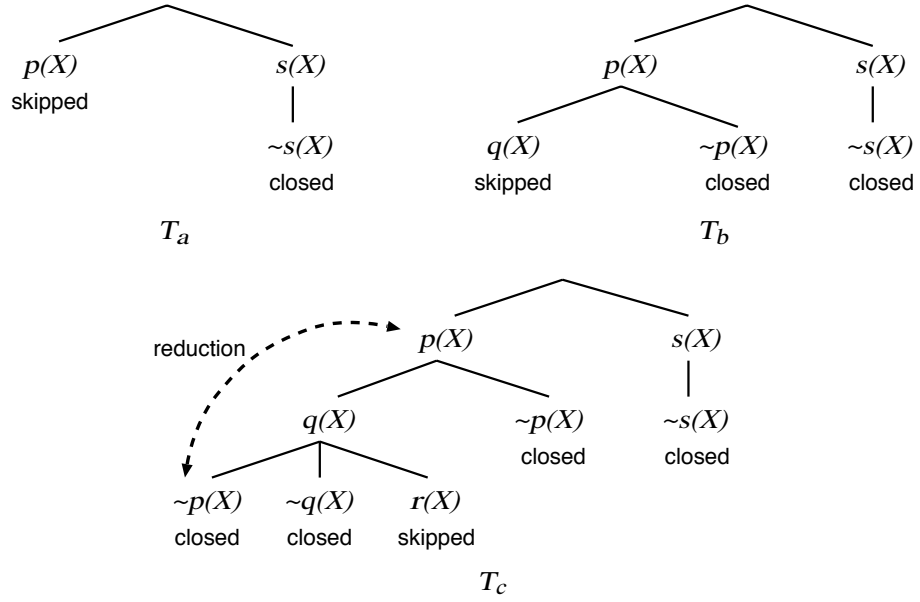


Figure 1: Solved tableaux of Example 2

is extended by using the unit clause $\neg s(Y)$. *Note that* $s(X)$ *cannot be skipped since the production field* $\mathscr{P}$ *limits the maximum length of consequences to one. The derived consequence is* $skip(T_a) = \{p(X)\}$. T_b *shows the other derived tableau whose node* $p(X)$ *is extended by* $q(X) \vee \neg p(X)$, *and the node* $q(X)$ *is skipped. The consequence of* T_b *is* $skip(T_b) = \{q(X)\}$. *In* T_c, *the nodes* $p(X)$ *and* $q(X)$ *are extended by* $q(X) \vee \neg p(X)$ *and* $\neg p(Z) \vee \neg q(Z) \vee r(Z)$ *respectively. The bottom node* $\neg p(X)$ *is closed by reduction with the ancestor* $p(X)$, *and* $r(X)$ *is skipped. The consequence is* $skip(T_c) = \{r(X)\}$. *There are some redundant solved tableaux other than* T_a, T_b *and* T_c. *The consequences of those are subsumed by* $p(X)$, $q(X)$ *or* $r(X)$[1]. *As the results, we can get three new characteristic clauses in this example:* $Newcarc(\Sigma, C, \mathscr{P}) = \{p(X), q(X), r(X)\}$.

3 Pruning Methods for SOL Tableau Calculus

3.1 Local Failure Caching for Length Condition

Local failure caching [9, 10] is an excellent pruning method for refutation finding which can avoid solving repetitiously a subgoal with the same or a more specific substitution. Iwanuma *et al*. [5] reformulated the local failure caching for consequence finding. This procedure is complete if a production field does not imply a maximum length condition. We review the procedure and then show a counter example. We first define the *SOL search tree* to explicitly express all possible SOL-deductions.

Definition 6. *The SOL search tree from* $\Sigma + C$ *and* $\mathscr{P}$ *via* ϕ *is a tree* $\mathscr{T}$ *labeled with tableaux as follows. We identify a node with its label (i.e., a tableau) if no confusion arises. The root of* $\mathscr{T}$ *is a tableau which consists of the start clause C only. Every non-leaf node T in* $\mathscr{T}$ *has as many successor nodes as there are successful applications of a single inference step applied to the selected subgoal in T by* ϕ, *and the successor nodes of T are the respective resulting tableaux. A segment of* $\mathscr{T}$ *is a subtree that contains the nodes explored by* ϕ *from the root to some node.*

[1]The new pruning method *skip-minimality* proposed in Section 3.2 can prune such redundant tableaux.

Definition 7 (Solution and Failure Substitution). *Given a SOL search tree $\mathscr{T}$ and a depth-first selection function. Let T be a tableau in $\mathscr{T}$ and K the selected subgoal in T.*

1. *If T' in $\mathscr{T}$ is a descendant tableau of T such that all branches through K in T' are solved, then the composition $\sigma = \sigma_1 \cdots \sigma_k$ of substitutions applied to the tableau T on the way from T to T' is called a solution of K at T via T'.*

2. *If $\mathscr{T}'$ is an initial segment of $\mathscr{T}$ containing no proof at T' or below it, then the solution σ is named a failure substitution for K at T in $\mathscr{T}'$.*

Definition 8 (Local Failure Caching Procedure for Consequence Finding [5]). *Let $\mathscr{T}'$ be a finite initial segment of a tableau search tree $\mathscr{T}$.*

Step 1: Whenever a subgoal K in a tableau T in $\mathscr{T}'$ has been solved via a tableau T', then the computed solution σ is stored at the node K.

   a. If T' cannot be completed to a solved tableau in $\mathscr{T}'$ and the proof procedure backtracks over T', then σ is turned into a failure substitution.

   b. If T' once has been completed in $\mathscr{T}'$ at a previous stage and the proof procedure backtracks over T' for searching alternative consequences, then continue the backtracking, without adding σ to the failure substitutions.

Step 2: In any alternative solution process of the subgoal K below the search node T, if a substitution $\tau = \tau_1 \cdots \tau_m$ is computed such that one of the failure substitutions stored at the node K is more general than τ, then the proof procedure immediately backtracks.

Step 3: When the search node T (at which K was selected) is backtracked, then all failure substitutions at K are deleted.

We show a counter example for the completeness of the above procedure.

Example 3. *We define a start clause $C = p(X) \vee q(X) \vee r(X)$, a set of clauses $\Sigma = \{ \neg q(X) \vee s(X),\ \neg s(Y) \}$, and a production field $\mathscr{P} = \langle \mathscr{L}^+,$ length is less than $3 \rangle$. We consider finding consequences from this problem. We can apply the Skip operation to $p(X)$ and $q(X)$ in the start clause since they are positive. But we cannot close $r(X)$, because the maximum length of consequences is limited to two (see T_1 in Figure 2). In this case, the local failure caching procedure stores an empty failure substitution $\sigma = \emptyset$ at the node $q(X)$. Since $\emptyset$ is the most general substitution, all the other applicable operations to $q(X)$ are pruned immediately. As the result, we cannot solve this problem and get any consequences. However, if we do not use the local failure caching, we can obtain the consequence $p(X) \vee r(X)$ from the solved tableau T_2 in Figure 2. Hence, the local failure caching is incomplete if there exists a maximum length condition.*

The cause of incompleteness is that the procedure does not consider skipped nodes. We extend the definitions of solution and failure substitution, and propose a complete procedure, called *local failure caching for length condition*.

Definition 9 (Extended Solution and Failure Substitution). *Given a tableau search tree $\mathscr{T}$ and a depth-first selection function. Let T be a tableau in $\mathscr{T}$ and K the selected subgoal in T.*

1. *If T' in $\mathscr{T}$ is a descendant tableau of T such that all branches through K in T' are solved, then the pair $\langle \sigma, s \rangle$, where σ is the composition $\sigma = \sigma_1 \cdots \sigma_k$ of substitutions applied to the tableau T on the way from T to T' and $s = skip(T')$, is called an extended solution of K at T via T'.*

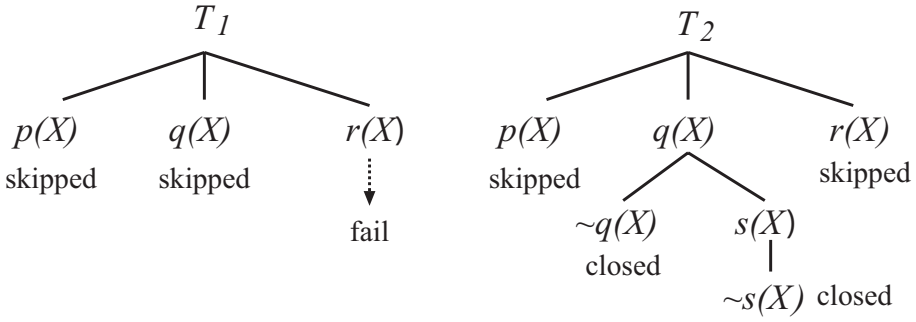


Figure 2: A counter example of local failure caching for consequence finding

2. *If $\mathscr{T}'$ is an initial segment of $\mathscr{T}$ containing no proof at T' or below it, then the extended solution $\langle\sigma, s\rangle$ is named an extended failure substitution for K at T in $\mathscr{T}'$.*

For two extended failure substitutions $\langle\sigma, s_1\rangle$ and $\langle\tau, s_2\rangle$, $\langle\sigma, s_1\rangle$ is more general than $\langle\tau, s_2\rangle$ if $\exists\theta(\sigma\theta = \tau \wedge s_1\theta \subseteq s_2)$, that is, σ is more general than τ and s_1 subsumes s_2. If $\mathscr{P}$ does not have a maximum length condition, then we simply ignore the latter condition, that is, $\langle\sigma, s_1\rangle$ is more general than $\langle\tau, s_2\rangle$ if $\exists\theta(\sigma\theta = \tau)$.

Definition 10 (Local Failure Caching Procedure for Length Condition)**.** *The definition is same as Definition 8 except for replacing terms of "solution" and "failure substitution" with "extended solution" and "extended failure substitution" respectively.*

The completeness of the local failure caching for length condition is given by the following proposition.

Proposition 1. *Let $\mathscr{T}$ be a tableau search tree, T a tableau in $\mathscr{T}$, K the selected subgoal in T, and T_a, T_b descendant tableaux of T. Suppose that $\langle\sigma, s_1\rangle$ is the extended failure substitution for K at T via T_a, and $\langle\tau, s_2\rangle$ is the extended solution of K at T via T_b generated by an alternative process. If T_b has a solved tableau T_{bs} as a descendant node in $\mathscr{T}$, then $\langle\sigma, s_1\rangle$ is not more general than $\langle\tau, s_2\rangle$.*

Proof. We prove the completeness by a reductio-ad-absurdum. We assume that $\langle\sigma, s_1\rangle$ is more general than $\langle\tau, s_2\rangle$, that is, $\exists\theta(\sigma\theta = \tau \wedge s_1\theta \subseteq s_2)$. Let S_a and S_{bs} be the subtableaux with root K in T_a and T_{bs} respectively (see Figure 3). Then, we replace S_{bs} in T_{bs} with $S_a\theta$, and denote the resulting tableau as T'_{bs}, which is solved and satisfies the maximum length condition. The reason is as follows. Let $\mathscr{N}$ be the set of selected subgoals on the way to T_{bs} after T_b. We extract the set $\mathscr{N}_f$ of nodes from $\mathscr{N}$ which are skipped by the Skip-factoring operation. There might exist the skipped nodes in $\mathscr{N}_f$ which have the factoring target in S_{bs} but not in $S_a\theta$. However, the number of such nodes is at most $|s_2 \setminus s_1\theta|$ since $s_1\theta \subseteq s_2$. Therefore, we can apply the Skip operation instead of the Skip-factoring to such nodes. As the results, $skip(T'_{bs})$ satisfies the maximum length condition.

This means that if the subtableau S_a was solved with the substitution $\sigma\theta$ instead of σ, then there should exist a solved tableau in $\mathscr{T}$ below T_a. Hence, if S_a is solved with the more general substitution σ, then there must be a solved tableau in $\mathscr{T}$ below T_a. But this contradicts the assumption of $\langle\sigma, s_1\rangle$ being a failure substitution. □

We consider solving Example 3 by using local failure caching for length condition. Since we cannot close $r(X)$ in T_1, the extended failure substitution $\langle\emptyset, \{p(X), q(X)\}\rangle$ is stored to the node $q(X)$. When the alternative process solves $q(X)$ using two extension operations (see T_2 in Figure 2), the extended solution substitution $\langle\emptyset, \{p(X)\}\rangle$ is stored to $q(X)$. Now, $\langle\emptyset, \{p(X), q(X)\}\rangle$ is not more general than $\langle\emptyset, \{p(X)\}\rangle$

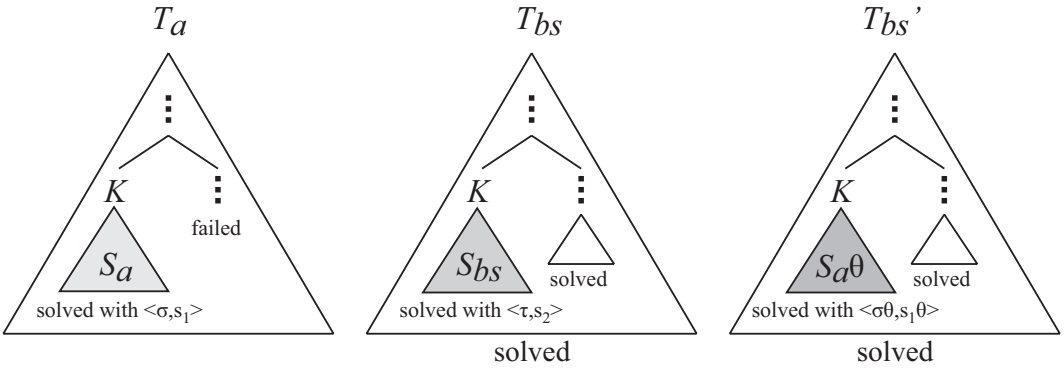


Figure 3: Completeness of local failure caching for length condition

because $\{p(X), q(X)\} \not\subseteq \{p(X)\}$. Hence, we can obtain the solved tableau $\mathcal{T}_2$ by skipping $r(X)$ and get the consequence $p(X) \vee r(X)$.

Since the Skip operation never instantiates a tableau, local failure caching is significantly effective for the SOL tableau calculus. Moreover, using the maximum length condition is important for finding simple and short consequences in practical time. In such a situation, local failure caching for length condition is an essential technique for practical use.

3.2 Skip-minimality

The task of consequence finding is to find all minimal consequences with respect to subsumption. This means that even if we find a solved tableau, we have to continue searching other solved tableaux for finding all minimal consequences. In proof finding, we can halt the computational process immediately if one refutation is found. This is an important difference between consequence finding and proof finding. Of course, if a refutation is found in a consequence finding process, then we can stop the computation immediately since the refutation is the most general consequence.

Skip-minimality improves the efficiency of such a consequence enumeration process. This method can prune unsolved tableaux which will generate only non-minimal consequences with respect to subsumption by using the consequences derived from already solved tableaux.

Definition 11 (Skip-Minimality)**.** *Let* Γ *be a set of clauses. A tableau* T *is skip-minimal for* Γ *if the clause* $skip(T)$ *is not properly subsumed by any clause in* Γ.

Suppose that Γ is a set of consequences derived by some SOL-deductions in a consequence enumeration process. If a tableau T is not skip-minimal for Γ, then T and all its descendant tableaux can be pruned immediately. Skip-minimality is complete since the four kinds of operations in Definition5 never generalize $skip(T)$. If we find general and short consequences in early stages, then skip-minimality is very effective since it prevents generating many redundant tableaux.

4 Search Strategy

Limiting the maximum length of consequences is important. When there is no limit, the branching factor of a SOL search tree increases and the size of the tree grows exponentially. However, it is difficult to know the appropriate length condition in advance. One of the solutions is to execute a search process with incrementing the maximum length limit iteratively, like DFID (*depth-first iterative deepening*) search strategy.

We propose the *consequence iterative lengthening* (CIL) strategy. Let P be a consequence enumeration process and l a length limit of consequences (initially $l = 0$). CIL strategy executes the process P with the length limit l repeatedly, incrementing l for each iteration. If l reaches a given limit or satisfies the condition $s < l$, then the iteration stops, where s is the maximum number of used Skip operations in every tableau.

Since CIL strategy visits the same tableaux in a SOL search tree multiple times, it may seem wasteful. However, it turns out to be not so costly, because the number of tableaux in the SOL search tree increases exponentially in proportion to incrementing the length limit of consequences. We show the property of CIL strategy in Section 5 experimentally.

5 Experimental Results

We have implemented skip-minimality, local failure caching for length condition and CIL strategy in SOLAR [12] which is a Java implementation of the SOL tableau calculus. Table 1 shows the experimental results for our proposed methods. We used some problems in the TPTP library v2.5.0 [14] as consequence finding problems, which are satisfiable instances (for example, "BOO008-3.p" is the file name of the problem). There are 1,092 satisfiable problems in the library and 24 categories contain such satisfiable instances. We selected 10 categories and chose one satisfiable problem from each category. We defined a production field in each problem as $\langle \mathscr{L},\ \text{length} \leq x \rangle$, where $\mathscr{L}$ is the set of all literals and the length condition is given as "len $\leq x$" in Table 1. "dep $\leq y$" means that we limited the maximum depth of tableaux to y. "sr", "sm" and "lfc" denote skip-regularity, skip-minimality and local failure caching for length condition respectively. "#C." and "#Steps" are the number of found consequences and the total number of derived tableaux (that is the number of nodes in a SOL search tree) respectively. The experiments were done on a Core Duo (1.66GHz) machine with 2GB memory. "t.o." means that the problem could not solve within 600 CPU seconds.

Table 1 shows that our proposed pruning methods have a great ability for reducing the search space and improving the speed. In particular, skip-minimality is very effective for solving GRP123-1.005. If we do not use skip-minimality to solve the problem, then we cannot solve the problem within 600 CPU seconds. In MSC009-1 and NLP026-1, local failure caching for length condition shows a high pruning effect. In BOO008-3 and LCL168-1, skip-minimality is not effective because the current implementation of clause-subsumption checking in SOLAR is naive. The improvement of the clause-subsumption check algorithm is one of the important future work. The rightmost row of Table 1 indicates that CIL strategy has almost no overhead. For example, in KRS005-1, CIL executes DFID five times with the maximum length limit of consequences from 0 to 4. However, the number of inferences and execution time are not so different than DFID without CIL.

6 Conclusion and Future Work

We have proposed new complete pruning methods and a search strategy for SOL tableau calculus. Local failure caching for length condition and skip-minimality can avoid producing many redundant tableaux. CIL is helpful to a user for finding short and simple consequences preferentially in a limited time. The completeness of local failure caching for length condition supports CIL strategy, and the pruning power of skip-minimality is promoted by CIL since it generates short consequences in early stages.

Currently, SOLAR does not have a special mechanism for handling equality. In order to solve a problem with equality efficiently, the development of the complete equality handling mechanism for consequence finding is one of the important future work.

Table 1: Experimental results

Problem	Params	Pruning Methods	#C	DFID		DFID + CIL	
				#Steps	Time [sec]	#Steps	Time [sec]
BOO008-3.p	dep ≤ 3 len ≤ 5	none	831	308,437	2.3	355,371	2.4
		sr	831	303,424	2.3	350,358	2.5
		sm	831	308,437	7.2	355,371	7.4
		lfc	831	299,195	2.4	342,711	2.5
		all	831	295,944	7.3	339,460	7.7
GRP123-1.005.p	dep ≤ 5 len ≤ 1	none	-	-	t.o.	-	t.o.
		sr	-	-	t.o.	-	t.o.
		sm	65	822,283	4.5	935,057	4.9
		lfc	-	-	t.o.	-	t.o.
		all	65	784,888	4.8	897,662	5.2
HWV034-1.p	dep ≤ 10 len ≤ 3	none	4	7,187,849	18.3	7,409,286	18.8
		sr	4	4,392,088	11.6	4,571,153	12.1
		sm	4	6,686,239	17.6	6,907,676	18.1
		lfc	4	4,733,454	14.2	4,952,443	15.1
		all	4	3,112,295	10.6	3,288,912	11.0
KRS005-1.p	dep ≤ 4 len ≤ 4	none	131	160,767,095	438.5	165,203,316	441.4
		sr	131	44,631,879	136.2	46,621,089	136.4
		sm	131	46,303,085	139.3	47,444,019	141.5
		lfc	131	106,297,270	319.2	110,027,318	333.5
		all	131	8,730,512	36.8	9,166,063	37.9
LCL168-1.p	dep ≤ 5 len ≤ 1	none	2,091	39,103	33.3	40,752	33.4
		sr	2,091	38,615	33.5	40,264	33.6
		sm	2,091	39,103	35.4	40,752	35.6
		lfc	2,091	38,629	33.4	40,278	33.8
		all	2,091	38,141	35.5	39,790	36.0
MSC009-1.p	dep ≤ 4 len ≤ 4	none	43	11,160,097	23.4	11,442,019	24.7
		sr	43	7,453,146	16.7	7,705,908	17.0
		sm	43	8,835,505	20.1	9,117,427	20.4
		lfc	43	4,786,900	12.0	4,915,009	12.2
		all	43	1,992,519	6.4	2,107,194	6.7
NLP026-1.p	dep ≤ 5 len ≤ 2	none	15	129,692,801	301.8	130,198,526	307.0
		sr	15	72,182,022	178.0	72,667,948	176.5
		sm	15	129,025,421	309.8	129,531,146	305.0
		lfc	15	8,218,827	23.7	8,716,795	25.6
		all	15	6,391,292	20.8	6,870,050	22.8
PUZ001-3.p	dep ≤ 10 len ≤ 3	none	26	34,997,018	134.7	39,946,809	145.8
		sr	26	14,845,990	59.6	17,823,280	66.4
		sm	26	731,711	3.8	1,157,985	5.0
		lfc	26	33,799,877	143.4	38,495,682	154.7
		all	26	458,832	3.2	759,780	4.3
SET777-1.p	dep ≤ 6 len ≤ 2	none	3	3,906,678	9.3	3,994,168	9.6
		sr	3	1,118,100	3.4	1,158,829	3.5
		sm	3	3,502,145	8.5	3,583,800	8.7
		lfc	3	2,251,658	7.1	2,302,211	7.3
		all	3	656,933	2.8	680,619	2.9
SYN084-1.p	dep ≤ 3 len ≤ 4	none	5	1,335,866	7.0	1,622,436	8.0
		sr	5	1,210,436	6.9	1,486,172	7.8
		sm	5	328,239	2.5	441,974	2.7
		lfc	5	1,335,866	7.3	1,621,274	8.2
		all	5	285,808	2.4	395,557	2.7

Ray and Inoue [13] have proposed a transformation method for unstable production fields, which converts these into stable ones. This method greatly helps to define a desired production field and to prune an unnecessary search space.

References

[1] K Inoue. Linear resolution for consequence finding. *Artificial Intelligence*, 56:301–353, 1992.

[2] Katsumi Inoue. Automated abduction. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, volume 2408 of *Lecture Notes in Computer Science*, pages 311–341. Springer, 2002.

[3] Katsumi Inoue. Induction as consequence finding. *Machine Learning*, 55(2):109–135, 2004.

[4] Katsumi Inoue, Koji Iwanuma, and Hidetomo Nabeshima. Consequence finding and computing answers with defaults. *Journal of Intelligent Information Systems*, 26(1):41–58, 2006.

[5] K Iwanuma, K Inoue, and K Satoh. Completeness of pruning methods for consequence finding procedure SOL. In *Proceedings of FTP-2000*, pages 89–100, 2000.

[6] Koji Iwanuma and Katsumi Inoue. Conditional answer computation in SOL as speculative computation in multi-agent environments. In *Proceedings of the Third International Workshop on Computational Logic in Multi-Agent Systems (CLIMA-02)*, pages 149–162, 2002.

[7] Koji Iwanuma and Katsumi Inoue. Minimal answer computation and SOL. In *Proceedings of the Eighth European Conference on Logics in Artificial Intelligence (JELIA 2002)*, volume 2424 of *Lecture Notes in Artificial Intelligence*, pages 245–257. Springer, 2002.

[8] Char Tung Lee. *A completeness theorem and a computer program for finding theorems derivable from given axioms*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, 1967.

[9] R. Letz, C. Goller, and K. Mayr. Controlled integration of the cut rule into connection tableau calculi. *Journal of Automated Reasoning*, 13(3):297–338, 1994.

[10] Reinhold Letz. Clausal tableaux. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction - A Basis for Applications*, volume I: Foundations, pages 39–68. Kluwer, Dordrecht, 1998.

[11] Donald W. Loveland. *Automated Theorem Proving: a logical basis*. North-Holland Publishing Company, Amsterdam, 1978.

[12] Hidetomo Nabeshima, Koji Iwanuma, and Katsumi Inoue. SOLAR: A consequence finding system for advanced reasoning. In *Proceedings of Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2003)*, pages 257–263, 2003.

[13] Oliver Ray and Katsumi Inoue. A consequence finding approach for full clausal abduction. In *Proceedings of the 10th International Conference on Discovery Science (DS 2007)*, pages 173–184, 2007.

[14] Geoff Sutcliffe and Christian Suttner. The TPTP problem library for automated theorem proving v2.5.0. http://www.tptp.org/, 2002.

Proofs and Refutations, and Z3

Leonardo de Moura and Nikolaj Bjørner
Microsoft Research

Abstract

Z3 [3] is a state-of-the-art Satisfiability Modulo Theories (SMT) solver freely available from Microsoft Research. It solves the decision problem for quantifier-free formulas with respect to combinations of theories, such as arithmetic, bit-vectors, arrays, and uninterpreted functions. Z3 is used in various software analysis and test-case generation projects at Microsoft Research and elsewhere. The requirements from the user-base range from establishing validity, dually unsatisfiability, of first-order formulas; to identify invalid, dually satisfiable, formulas. In both cases, there is often a need for more than just a yes/no answer from the prover. A model can exhibit why an invalid formula is not provable, and a proof-object can certify the validity of a formula. This paper describes the proof-producing internals of Z3. We also briefly introduce the model-producing facilities. We emphasize two features that can be of general interest: (1) we introduce a notion of implicit quotation to avoid introducing auxiliary variables, it simplifies the creation of proof objects considerably; (2) we produce natural deduction style proofs to facilitate modular proof re-construction.

1 Introduction

The title of our paper borrows from Imre Lakatos's famous book on conjectures, proofs and refutations in informal mathematics [7], yet our setting is machine checked proofs, that are penultimately given in a context of formal systems where proofs are derived from axioms. Proofs in our context are derivations from axioms, or derivations in theories that have solvers, implemented using efficient algorithms that need only produce derivations implicitly. Part of the challenge is that efficiently and compact checkable proofs, or certificates, are to be extracted from the solvers. Refutations, also called models, are counter-examples, that exhibit interpretations for formulas that do not follow from asserted axioms. Models are also extracted from solvers.

Applications of SMT solvers that consume models benefit tremendously from a prover that produces more than just a yes/no answer or a set of saturated clauses. A solver should therefore be able to communicate a model that can be represented finitely and consumed by the clients. Applications that mainly require an indication of validity may in some cases furthermore benefit from a certificate in the form of a proof object. This paper describes the model-producing features of Z3, currently available in the tool. It also describes the proof-producing features in preparation in the next version of Z3. While this paper can serve as an overview of the model- and proof-producing facilities in Z3, we point out the following particularities of our approach:

1. We define a notion of implicit quotation that allows us to encode a Tseitsin' style clausification without introducing auxiliary symbols. Other proof-producing SMT systems that we are aware of [18], [19], [2], [22], introduce auxiliary symbols (such as proxy literals) during clausification and other transformations. Such symbols can impede optimizations in the theory solvers (we provide an example in Section 3.3.2 where we can avoid introducing extra Simplex Tableaux rows) and make proof re-construction more involved.

2. We adapt an open-ended architecture for representing proofs. Proof rules can be introduced by respective theory modules and combined with others. At the propositional level this is manifested

Rudnicki P, Sutcliffe G., Konev B., Schmidt R., Schulz S. (eds.);
Proceedings of the Combined KEAPPA - IWIL Workshops, pp. 123-132

as we adapt a natural deduction style calculus. This contrasts with existing proof-producing SAT solvers that generate resolution proofs directly [10, 6, 21]. We are obviously not the first to use natural deduction in the context of SMT, for example, [9], investigates efficient proof checking of natural deduction style proofs by implementing inference rules as rewrites.

3. We also do not attempt to specify all inference rules from a smaller set of axioms. Instead we rely on proof checking to be able to carry out a limited set of inferences, or refine proofs in a separate pass. This choice obviously reflects a trade-off between the requirements on the solver vs. the proof checker. Our own experience has been that the coarse granularity has in fact been sufficient in order to catch implementation bugs. Future work includes investigating whether this approach is practical in the context of proof checkers based on trusted cores [1].

2 Preliminaries

2.1 Terms and Formulas

Z3 uses basic multi-sorted first-order terms. Formulas are just terms of Boolean sort, and terms are built by function application, quantification, and bound variables. Sorts range over a finite denumerable set of disjoint primitive sorts. To summarize

$$\begin{array}{lllll} s \in \mathit{Sorts} & ::= & \mathsf{Boolean} \mid \mathsf{Int} \mid \mathsf{Proof} \mid \ldots & \\ t \in \mathit{Terms} & ::= & f(t_1,\ldots,t_n) & \text{function application} \\ & \mid & x & \text{bound variable} \\ & \mid & \forall x : s \,.\, t \mid \exists x : s \,.\, t & \text{quantification} \end{array}$$

There a a few built-in sorts, such as Boolean, Int, Real, BitVec[n] (for each n, an n-bit bit-vector), and Proof. The Proof sort is used for proof-terms. Terms can be annotated by pragmas. For example, quantifiers are annotated with patterns that control quantifier instantiation. Function symbols can be both interpreted and uninterpreted. For example, numerals are encoded using interpreted functions. Function symbols also have attributes, such as to indicate whether they are associative and/or commutative. Note that there are no binding operators other than universal and existential quantification. A number of function symbols are built-in to the base theory. We will introduce the set of proof-constructing terms as we explain their origination, but here let us summarize the main pre-declared function symbols. We use the usual infix symbols $\wedge, \vee, \rightarrow, \leftrightarrow$ for Boolean conjunction, disjunction, implication and bi-implication. For each sort s there is an equality relation $\simeq$: $s \times s \rightarrow$ Boolean. The relation $\sim$: Boolean $\times$ Boolean $\rightarrow$ Boolean is used in proof terms. A proof of $\varphi \sim \psi$ establishes that φ is equisatisfiable with ψ. In other words, if we close φ and ψ by second-order existential quantification of all Skolem functions and constants, we obtain equivalent formulas.

2.2 Proof Terms

Proof objects are also represented as terms. So a proof-tree is just a term where each inference rule is represented by a function symbol. For example, consider the proof-rule for modus ponens:

$$\mathsf{modus\_ponens}\ \frac{\begin{array}{cc} \vdots p & \vdots q \\ \psi \rightarrow \varphi & \psi \end{array}}{\varphi}$$

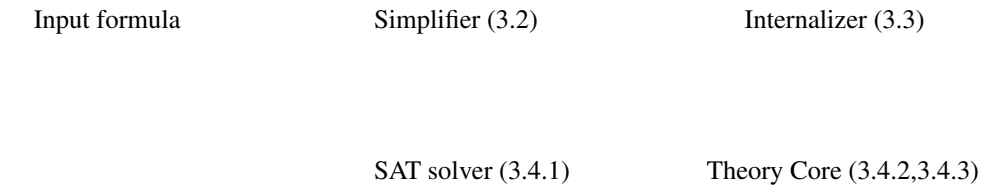
In the rule, p is a proof-term for $\psi \rightarrow \varphi$, and q is a proof for ψ. The resulting proof-term for φ is then $mp(p,q,\varphi)$. We will later elaborate on the function symbols for building basic proof terms that are available in Z3.

Every proof term has a *consequent*, the formula that the proof establishes. It is always the last argument in our proof terms. To access the consequent we will use the notation $con(p)$. For example, $con(mp(p,q,\varphi)) = \varphi$.

3 Proofs

3.1 Overview

Z3 applies multiple stages when attempting to verify a formula. First formulas are simplified using a repository of simplification rules. The result of simplification is then converted into an internal format that can be processed by the solver core. We refer to this phase as *internalization*. The core comprises of a SAT solver that performs Boolean search and a collection of theory solvers. The solver for equality and uninterpreted function symbols (congruence closure) features predominantly as a theory solver in Z3 as it dispatches constraints between the SAT solver and other theories.



As the figure illustrates, we will describe the various modules in the following sections, as we introduce the proof terms that are produced as a side effect of the modules.

The figure does not reflect the full extent of Z3. We will not be elaborating on proof objects for non-ground formulas in this paper. Z3 does produce proof objects for non-ground formulas. The required machinery introduces judgments stating equi-satisfiability of formulas. Furthermore, Z3 contains a module that produces and integrates proofs in a superposition calculus [4]. The proof terms are the usual superposition inferences [12]. That material is beyond the scope of this paper.

3.2 Simplification Rewriting

In a first phase, formulas are simplified using a rewriting simplifier. The simplifier applies standard simplification rules for the supported theories. For example, terms using the arithmetical operations, both for integer, real, and bit-vector arithmetic, are normalized into sums of monomials. A single axiom called *rewrite* is used to record the simplification steps.

rewrite$(t \simeq s)$, *rewrite*$(\varphi \leftrightarrow \psi)$ A proof for a local rewriting step that converts t to s or φ to ψ. The head function symbol of t is interpreted. Sample instances of this proof object are: *rewrite*$(x+0 \simeq x)$, *rewrite*$(x+x \simeq 2\cdot x)$, and *rewrite*$((\varphi \vee \texttt{false}) \leftrightarrow \varphi)$.

Notice that we do not axiomatize the legal rewrites. Instead, to check the rewrite steps, we rely on a proof checker to be able to apply similar inferences for the set of built-in theories: arithmetic, bit-vectors and arrays.

3.3 Internalization

Internalization is the process of translating an arbitrary formula φ into a normal form that can be consumed by efficient proof-search procedures. We will discuss two internalizations: clausification and conversion of arithmetical constraints into a format that can be processed using a global Simplex tableau. Internalization often introduces auxiliary variables that are satisfiability preserving definitional extensions of the original problem. We will introduce a simple, but apparently unrecognized, technique of *implicit quotation* to allow us introducing these definitional extensions, but at the same time take advantage of the fact that the auxiliary variables can be viewed directly as terms they are shorthand for. The technique of implicit quotation makes the proof extraction process fairly direct.

3.3.1 Clausification

Tseitsin's clausal form conversion algorithm can be formulated as a procedure that works by recursive descent on a formula and produces a set of equi-satisfiable set of clauses. A simple conversion algorithm introduces one fresh name for each sub-formula, and defines the name using the shape of the sub-formula. We here recall the basic idea by converting and-or formulas into CNF.

$$
\begin{array}{lcll}
\mathit{cnf}(\varphi) & = & \textbf{let } (\ell, F) = \mathit{cnf}'(\varphi) \textbf{ in } \ell \wedge F & \\
\mathit{cnf}'(\ell) & = & (\ell, \mathit{true}) & \ell \text{ is a literal} \\
\mathit{cnf}'(\neg\varphi) & = & \textbf{let } (\ell, F) = \mathit{cnf}'(\varphi) \textbf{ in } (\neg\ell, F) & \\
\mathit{cnf}'(\varphi \wedge \psi) & = & \textbf{let } (\ell_1, F_1) = \mathit{cnf}'(\varphi),\ (\ell_2, F_2) = \mathit{cnf}'(\psi) \textbf{ in} & \\
 & & (p, F_1 \wedge F_2 \wedge (\neg\ell_1 \vee \neg\ell_2 \vee p) \wedge (\neg p \vee \ell_1) \wedge (\neg p \vee \ell_2)) & p \text{ is fresh} \\
\mathit{cnf}'(\varphi \vee \psi) & = & \textbf{let } (\ell_1, F_1) = \mathit{cnf}'(\varphi),\ (\ell_2, F_2) = \mathit{cnf}'(\psi) \textbf{ in} & \\
 & & (p, F_1 \wedge F_2 \wedge (\ell_1 \vee \ell_2 \vee \neg p) \wedge (p \vee \neg\ell_1) \wedge (p \vee \neg\ell_2)) & p \text{ is fresh}
\end{array}
$$

More sophisticated CNF conversions that do not introduce fresh names for all sub-formulas exist [14]. They control the number of auxiliary literals and clauses introduced during clausification.

Z3 does not introduce auxiliary predicates during internalization of quantifier-free formulas. Instead, it re-uses the terms that are already used for representing the sub-formulas. Thus, instead of introducing a fresh variable p, in the CNF conversion of $\varphi \vee \psi$, we treat the term $\varphi \vee \psi$ as a *literal*. To clarify that a sub-formula plays the rôle of a literal below, we *quote* it. So the literal associated with $\varphi \vee \psi$ is $\lceil \varphi \vee \psi \rceil$. So the CNF conversion of $\varphi \vee \psi$ produces the pair:

$$(\lceil \varphi \vee \psi \rceil,\ F_1 \wedge F_2 \wedge (\ell_1 \vee \ell_2 \vee \neg\lceil \varphi \vee \psi \rceil) \wedge (\lceil \varphi \vee \psi \rceil \vee \neg\ell_1) \wedge (\lceil \varphi \vee \psi \rceil \vee \neg\ell_2))$$

It may appear that we need to justify the auxiliary clauses $(\ell_1 \vee \ell_2 \vee \neg\lceil \varphi \vee \psi \rceil)$, $(\lceil \varphi \vee \psi \rceil \vee \neg\ell_1)$, and $(\lceil \varphi \vee \psi \rceil \vee \neg\ell_2)$ by appealing to the equi-satisfiablility, but the justification for these clauses happens in fact to directly use equivalence. First note that it follows by induction on the clausification algorithm that ℓ_1 is equivalent to φ and ℓ_2 is equivalent to ψ. Each of the auxiliary clauses introduced during clausification is therefore justified as propositional tautologies. Only limited propositional reasoning is required to justify these. The proof terms corresponding to the auxiliary clauses are tagged as definitional axioms. For example, the axiom $\lceil \varphi \vee \psi \rceil \vee \neg\varphi$ is represented by the term $\mathit{def\_axiom}((\varphi \vee \psi) \vee \neg\varphi)$.

We introduced quotation here to clarify in which context logical connectives were to be used. Our implementation in Z3 does not use quotation at all.

3.3.2 Arithmetic

Implicit quotation is also used when introducing auxiliary variables for theories, such as the theory for linear arithmetic. Let us recall how the Simplex solver in Z3 works. Following [5], a theory solver for

linear arithmetic, and integer linear arithmetic can be based on a Simplex Tableau of the form:

$$x_i \simeq \sum_{x_j \in \mathscr{N}} a_{ij} x_j \quad x_i \in \mathscr{B}, \tag{1}$$

where $\mathscr{B}$ and $\mathscr{N}$ denote the set of basic and nonbasic variables, respectively. The sets $\mathscr{B}$ and $\mathscr{N}$ are assumed disjoint, and each basic variable occurs in exactly one row. Thus, the values of basic variables are determined by the values of the non-basic variables. In addition to this tableau, the solver state stores upper and lower bounds l_i and u_i for every variable x_i and a mapping β that assigns a rational value $\beta(x_i)$ to every variable x_i. The bounds on nonbasic variables are always satisfied by β, so the following invariant is maintained by the tableau operations

$$\forall x_j \in \mathscr{N},\ \ l_j \leq \beta(x_j) \leq u_j. \tag{2}$$

A tableau is *satisfied* if the same inequalities hold for the basic variables:

$$\forall x_j \in \mathscr{B},\ \ l_j \leq \beta(x_j) \leq u_j. \tag{3}$$

Bounds constraints for basic variables are not necessarily satisfied by β, so for instance, it may be the case that $l_i > \beta(x_i)$ for some basic variable x_i, but pivoting steps can be used to fix bounds violations, or detect an unsatisfiable tableau.

Formally, Simplex-based solvers for linear arithmetic can interface with Boolean combinations of inequalities by introducing one slack variable and a tableau row for every (maximal) linear arithmetic sub-term in a formula. For example, it is by now common for SMT solvers to transform problems of the form:

$$[(x+y>2) \wedge (2x-y<2)] \vee [(f(z+x) \geq 5) \wedge z<3] \tag{4}$$

to the following equi-satisfiable formula:

$$\begin{aligned} & [(s_1>2) \wedge (s_2<2)] \vee [(f(s_3) \geq 5) \wedge z<3] \\ \wedge \quad & s_1 \simeq x+y \wedge s_2 \simeq 2x-y \wedge s_3 \simeq z+x \end{aligned} \tag{5}$$

The definitions for the slack variables translate into rows in a Simplex tableau. If we represent s_1 by the quotation $\lceil x+y \rceil$, and s_2 by $\lceil 2x-y \rceil$, and s_3 by $\lceil z+x \rceil$, we observe directly that the additional equalities are tautologies and therefore have trivial justifications. Furthermore, all Simplex tableau operations, including pivoting, are equivalence preserving (pivoting replaces equals for equals, and divides rows by constants), so the justifications for the Simplex rows remains a matter of expanding quotations and checking equalities using linear arithmetic. The above observation can be used to reconstruct proofs from unsatisfiable tableaux in a direct manner. In contrast, [2] proposed an encoding of arithmetical constraints by introducing slack variables for potentially every arithmetical subterm (including potentially a slack variable for z in $z<3$). The slack variables allowed for tracking explanations and extracting proofs from infeasible tableaux.

We now explain how proofs are extracted from unsatisfiable tableaux without modifying the translation phase. With a tableau row of the form (1) associate the suprema and infima of implied by the coefficients, namely, let a_r be the coefficients from the row and l_j and u_j be the lower and upper bounds that are asserted by the literals $x_j \leq u_j$ and $l_j \leq x_j$, then:

$$\mathsf{sup}(a_r) \quad := \quad \sum_{x_j \in \mathscr{N}^+} a_{rj} u_j + \sum_{x_j \in \mathscr{N}^-} a_{rj} l_j \tag{6}$$

$$\mathsf{inf}(a_r) \quad := \quad \sum_{x_j \in \mathscr{N}^+} a_{rj} l_j + \sum_{x_j \in \mathscr{N}^-} a_{rj} u_j \tag{7}$$

where $\mathscr{N}^- = \{x_j \mid a_{rj} < 0\}$, and $\mathscr{N}^+ = \{x_j \mid a_{rj} > 0\}$; and as usual, we set $\mathsf{sup}(a_r) = \infty$ if either some $x_j \in \mathscr{N}^+$, $u_j = \infty$, or for some $x_j \in \mathscr{N}^-$, $l_j = -\infty$.

Then an unsatisfiable tableau can be identified by an infeasible row r, where

$$u_r < \mathsf{inf}(a_r) \text{ and } x_r \leq u_r \text{ is asserted, or } l_r > \mathsf{sup}(a_r) \text{ and } l_r \leq x_r \text{ is asserted.} \tag{8}$$

Corresponding to an infeasible row, we can extract a theory conflict by accumulating the bounds that were used to derive a contradiction. So for example, in case of $u_r < \mathsf{inf}(x_r)$, the conflict clause is of the form

$$\neg(x_r \leq u_r) \vee \bigvee_{x_j \in \mathscr{N}^+} \neg(l_j \leq x_j) \vee \bigvee_{x_j \in \mathscr{N}^-} \neg(x_j \leq u_j) \tag{9}$$

It is now simple to prove the conflict clause:

$$\dfrac{\dfrac{\dfrac{\dfrac{x_r \simeq \sum_{x_j \in \mathscr{N}} a_{rj} x_j \qquad l_{r1} \leq x_{r1},\ x_{r1} \in \mathscr{N}^+}{x_r \geq (\sum_{x_j \in \mathscr{N}} a_{rj} x_j) - a_{r1} x_{r1} + a_{r1} l_{r1}} \qquad x_{r2} \leq u_{r2},\ x_{r2} \in \mathscr{N}^-}{x_r \geq (\sum_{x_j \in \mathscr{N}} a_{rj} x_j) - a_{r1} x_{r1} + a_{r1} l_{r1} - a_{r2} x_{r2} + a_{r2} u_{r2}} \quad \ldots}{\dfrac{\ldots}{x_r \geq \mathsf{inf}(a_r)}} \qquad x_r \leq u_r}{\dfrac{\bot}{\neg(x_r \leq u_r) \vee \bigvee_{x_j \in \mathscr{N}^+} \neg(l_j \leq x_j) \vee \bigvee_{x_j \in \mathscr{N}^-} \neg(x_j \leq u_j)}\ (9)}\ \mathit{lemma}$$

The left-most antecedent is a tautology in the theory of linear arithmetic, the other antecedents are hypotheses. The *lemma* inference rule collects (see Section 3.4) the disjunction of the negated hypotheses used for deriving $\bot$. The intermediary inferences correspond to basic inequality propagation. Our implementation in Z3 only produces the theory lemma directly without listing the equality corresponding to the infeasible row.

3.4 Modular Proofs

A basic underlying principle for composing and building proofs in Z3 has been to support a modular architecture that works well with theory solvers that receive literal assignments from other solvers and produce contradictions or new literal assignments. The theory solvers should be able to produce independent and opaque explanations for their decisions.

Conceptually, each solver acts upon a set of hypotheses and produce a consequent. The basic proof-rules that support such an architecture can be summarized as: *hypothesis*, that allow introducing an assumption, *lemma*, that eliminates hypotheses, and *unit_resolution* that handles basic propagation. We say that a proof-term is *closed* when every path that ends with a hypothesis contains an application of rule *lemma*. If a term is not closed, it is *open*. To summarize, these core rules are:

hypothesis(φ) Mark φ as a hypothesis. The resulting proof term is open.

lemma$(p, \neg\varphi_1 \vee \ldots \vee \neg\varphi_n)$ The proof term has one antecedent p such that $con(p) = false$, but p is open with hypotheses $\varphi_1, \ldots, \varphi_n$. The resulting proof term is closed.

unit_resolution$(p_0, p_1, \ldots, p_n, \psi_1 \vee \ldots \vee \psi_m)$ Where $con(p_0) = \varphi_1 \vee \ldots \vee \varphi_n \vee \psi_1 \vee \ldots \vee \psi_m$, $con(p_1) = \neg\varphi_1, \ldots con(p_n) = \neg\varphi_n$.

We will next describe how these rules integrate within a DPLL(T) architecture.

3.4.1 Proofs from DPLL(T)

The propositional inference engine in Z3 is based on a DPLL(T) architecture. We refer to [13] for an exposition on a basic introduction on DPLL(T) as a transition system. The main points we will use is that DPLL(T) maintains a state of the form $M \| F$ during search, where M is a partial assignment of the atomic predicates in the formula F. Furthermore, we assume F is in conjunctive normal form. The search keeps assigning atoms in M based on unit propagation, *theory* propagation (for example $x > 3$ implies that $x > 0$ by the theory of arithmetic), and guesses (also called decisions) until either it reaches a state where the assignment satisfies all clauses in F, or some clause in F contradicts the assignment in M.

The DPLL(T) proof search method lends itself naturally to producing resolution style proofs. Systems, such as zChaff, and a version of MiniSAT [10, 6, 21], produce proof logs based on logging the unit propagation steps as well as the conflict resolution steps. The resulting log suffices to produce a propositional resolution proof. This approach works even though the SAT solver can choose to restart or garbage collect learned conflict clauses that were produced during search.

The approach taken in Z3 bypasses logging, and instead builds proof objects during conflict resolution. With each clause we attach a proof. Clauses that were produced as part of the input have proofs that were produced from the previous steps. A clause that is produced during conflict resolution depends on some state of the partial model M. In particular, the learned clause is contradictory with some subset of the decision literals in M, either directly because the learned clause contains decision literals, or because the learned clause contains a literal that was obtained by propagation. Given a conflict clause $C : \ell_1 \vee \ell_2 \vee \ldots \vee \ell_n$, we build a proof term of the form $\mathit{lemma}(p, \ell_1 \vee \ell_2 \vee \ldots \vee \ell_n)$, where p is constructed by examining the justifications for $\neg\ell_1, \ldots, \neg\ell_n$. If $\neg\ell_i$ is a decision literal, then the justification for $\neg\ell_i$ is a term $\mathit{hypothesis}(\neg\ell_i)$. If $\neg\ell_i$ was inferred by unit propagation (so there is a fact $\neg\ell_i$ in M, with justification $C \vee \neg\ell_i$), then it is proved using unit-resolution and the justification for the clause $C \vee \neg\ell_i$.

We see that this approach does not require logging resolution steps for every unit-propagation, but delays the analysis of which unit propagation steps are useful until conflict resolution. The approach also does not produce a resolution proof directly. It produces a natural deduction style proof with hypotheses.

Other propositional rules that are used during proof-reconstruction are:

$\mathit{asserted}(\varphi)$ The formula φ is a user-supplied assumption.

$\mathit{goal}(\varphi)$ The formula φ is a user-supplied goal. A goal is symmetric to *asserted*, but allows retaining the distinction between goals and assumptions in proof objects.

$\mathit{mp}(p, q, \varphi)$ Proof of φ by modus ponens. Assume that $\mathit{con}(p) = \psi$ and that $\mathit{con}(q)$ is either $\psi \rightarrow \varphi$ or $\psi \leftrightarrow \varphi$. The latter form is used extensively in the simplifier to apply equivalence-preserving simplification steps.

3.4.2 Congruence Proofs

In Z3, the congruence closure implementation is tightly integrated with the Boolean satisfiability core. It serves as a main hub for equality propagation. The efficient extraction of minimal justifications for congruence closure proofs has been studied extensively, [11]. We here summarize the proof objects that are extracted from the justifications.

The theory of equality can be captured by axioms for reflexivity, symmetry, transitivity, and substitutivity of equality. We encode these axioms as inference rules, and furthermore only specify that these inference rules apply for any binary relation that is reflexive, symmetric, transitive, and/or reflexive-monotone. We use the terminology, reflexive-monotone, for relations that are reflexive and monotone in

a given function symbol f. In particular, the relation $\sim$ (from Section 2.1) is also an equivalence relation, and reflexive-monotone over conjunction and disjunction. So the rules are:

refl$(R(t,t))$ A proof for $R(t,t)$, where R is a reflexive relation.

symm$(p,R(t,s))$ A proof of $R(t,s)$, where R is a symmetric relation, and $con(p) = R(s,t)$.

trans$(p,q,R(t,s))$ A proof of $R(t,s)$, where R is transitive, and $con(p) = R(t,u)$ and $con(q) = R(u,s)$.

monotonicity$(p_1,..,p_n,R(f(t_1,..,t_n),f(s_1,..,s_n)))$ A proof of $R(f(t_1,..,t_n),f(s_1,..,s_n))$, where $con(p_1) = R(t_1,s_1)$, ..., $con(p_n) = R(t_n,s_n)$, and R is reflexive and monotone in f. The antecedent p_i can be suppressed if $t_i = s_i$. That is, reflexivity proofs are suppressed to save space.

Our congruence closure core maintains a congruence table. A congruence table enables propagation of equalities over function symbols, so that for example if $f(s,t)$ is a term, and s' is equal to s, then when creating $f(s',t)$ it is detected that the potentially new term is in the same congruence class as $f(s,t)$. The implementation also treats equality as a function, and every equality is also inserted to the congruence table. This makes detecting implied dis-equalities simple: given two terms s and t, search the congruence table for an existing entry for $s \simeq t$. If the table contains such a literal, then check if the literal is assigned to *false*. To make this use of the congruence table effective it understands commutative operations, such as equality. This implicit use of commutativity gets reflected in the generated proof terms, and we include a special rule for commutative functions. It can be instantiated for equalities.

comm$(f(s,t) \simeq f(t,s))$, *comm*$(f(s,t) \leftrightarrow f(t,s))$ where f is a commutative function (relation).

3.4.3 Theory lemmas

In the DPLL(T) architecture, decision procedures for a theory T identify sets of asserted T-inconsistent literals. Dually, the disjunction of the negated literals are T-tautologies. Consequently, proof terms created by theories can be summarized using a single form, here called Theory lemmas.

th_lemma$(p_1,\ldots,p_n,\varphi)$ Generic proof rule for theory lemmas. The formula $con(p_1) \wedge \ldots \wedge con(p_n) \rightarrow \varphi$ should be a T-tautology.

3.5 Applications

Z3 with proofs is still under development and has not been released yet. An obvious current application is that proofs offer a simple litmus test on the implementation for soundness bugs. We integrate a simple, but partial (it does not check T-lemmas) proof-checker in Z3 for this purpose. A much more effective strategy for debugging bugs in theory solvers has been to dump the T-lemmas as they are produced. Similar to [20], we can then apply an independent solver (namely, our previous version of Z3) on the T-lemmas. We found this approach very effective in debugging optimizations that turned out to be unsound.

We also have a facility for displaying proof-terms, but the proof term visualization very easily becomes too large to be of any use.

In future applications, we envision applications of proofs in Z3, such as: Proof-mining [17]; can proofs be mined for strategies that are helpful for speeding up proofs for a class of problems? Interpolation. Proof visualization. Finally, in the context of Isabelle/HOL, it has been suggested [8] to translate HOL formulas (which use polymorphism), into first-order untyped formulas. A potentially unsound translation is then run through first-order provers, but the produced proofs (currently apparently only Prover9), can be checked for whether they use inferences that contradict the types.

3.6 The overhead of enabling proofs

We benchmarked proof generation on a few selected, but non-trivial examples from SMT-LIB. The samples show a memory overhead of between 3x-40x, and corresponding slowdowns of 1.1x to 3x.

Benchmark	Without Proofs		With Proofs	
NEQ016_size7	26.01 secs	9.1 MB	39.84 secs	426 MB
PEQ010_size8	6.65 secs	9.3 MB	7.63 secs	40 MB
fischer6-mutex-14	7.01 secs	14 MB	16.38 secs	142 MB
cache.inv16	15.99 secs	11 MB	24.61 secs	159 MB
xs_20_40	2.2 secs	5.8 MB	2.70 secs	15.5 MB

4 Models

We will very briefly summarize the model generation features in Z3. More material is available on-line on `http://research.microsoft.com/projects/z3/models.html`.

Z3 has the ability to produce models as part of its output. Models assign values to the constants in the input and generate finite or partial function graphs for predicates and function symbols.

A model comprises of a set of partitions, each partition is printed as: $*k$, where k is a non-negative number. Each partition is associated with a set of constants from the input, and is associated with a concrete value. A concrete value can be either a Boolean (*true* or *false*), a numeral (with type Int, Real, or BitVec[n]), an array (represented as a finite map), a tuple (represented by a constructor and sequence of values for the fields), or an an uninterpreted ur-element. Ur-elements are internally represented as natural number numerals, but with an uninterpreted type.

By default Z3, produces a full (and compact) interpretation for free functions. There is an option to force Z3 to not assign interpretations to functions when their values don't influence the truth assignment to the formula.

Z3 version 2 integrates a superposition theorem prover [4]. When it is able to finitely saturate the set of non-ground input clauses, it can also report that the non-ground formula that was provided is satisfiable, but there are no other mechanisms for extracting additional information from the saturated set of clauses.

4.1 Applications

Models are by now used in a number of Z3 clients. The main clients that use models are the program exploration and test-case generation tools Pex and SAGE (we refer to [3] for all pointers). They extract symbolic path conditions by monitoring program executions and use Z3 to find alternate inputs that can guide the next execution into a different branch. Models are also used for improved debugging feedback from Spec# and for iterative counter-example guided refinement in the context of bounded model checking of model programs.

We also believe that the availability of models can in future applications play a useful role in the context of model-based quantifier instantiation and integrating external decision procedures with Z3.

5 Conclusions

We presented the proof and model generation facilities in Z3. Models have already shown particular usefulness in the context of SMT applications. Proofs will be available in Z3 v2, and we hope, in light of this introduction, that users will be able to find useful applications of the feature.

References

[1] Aaron Stump and Duckki Oe. Towards an SMT Proof Format. In *6'th International Workshop on SMT*, 2008.

[2] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Efficient Interpolant Generation in Satisfiability Modulo Theories. In Ramakrishnan and Rehof [15], pages 397–412.

[3] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In Ramakrishnan and Rehof [15].

[4] Leonardo de Moura and Nikolaj Bjørner. Engineering DPLL(T) + Saturation. In *IJCAR'08*, 2008.

[5] Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV'06*, LNCS 4144, pages 81–94. Springer-Verlag, 2006.

[6] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[7] Imre Lakatos. *Proofs and Refutations*. Cambridge University Press, 1976.

[8] Jia Meng and Lawrence C. Paulson. Translating Higher-Order Clauses to First-Order Clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008.

[9] Michal Moskal. Rocket-Fast Proof Checking for SMT Solvers. In Ramakrishnan and Rehof [15], pages 486–500.

[10] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535. ACM, 2001.

[11] Robert Nieuwenhuis and Albert Oliveras. Proof-Producing Congruence Closure. In Jürgen Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 2005.

[12] Robert Nieuwenhuis and Albert Rubio. Paramodulation-Based Theorem Proving. In Robinson and Voronkov [16], pages 371–443.

[13] Roberto Niewenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006.

[14] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Robinson and Voronkov [16], pages 335–367.

[15] C. R. Ramakrishnan and Jakob Rehof, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*. Springer, 2008.

[16] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[17] S. Schulz. Learning Search Control Knowledge for Equational Theorem Proving. In F. Baader, G. Brewka, and T. Eiter, editors, *Proc. of the Joint German/Austrian Conference on Artificial Intelligence (KI-2001)*, volume 2174 of *LNAI*, pages 320–334. Springer, 2001.

[18] Aaron Stump, Clark W. Barrett, and David L. Dill. Producing proofs from an arithmetic decision procedure in elliptical lf. *Electr. Notes Theor. Comput. Sci.*, 70(2), 2002.

[19] Aaron Stump and David L. Dill. Faster Proof Checking in the Edinburgh Logical Framework. In Andrei Voronkov, editor, *CADE*, volume 2392 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2002.

[20] Geoff Sutcliffe. Semantic Derivation Verification: Techniques and Implementation. *International Journal on Artificial Intelligence Tools*, 15(6):1053–1070, 2006.

[21] Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic*, July 2007.

[22] Yeting Ge and Clark Barrett. Proof Translation and SMT-LIB Benchmark Certification: A Preliminary Report. In *6'th International Workshop on SMT*, 2008.