EPiC Series in Computing

Volume 44, 2017, Pages 46–60

Vampire 2016. Proceedings of the 3rd Vampire Workshop

Better Proof Output for Vampire

Giles Reger

University of Manchester, Manchester, UK

Abstract

Vampire produces highly usable and informative proofs, but now they are even better and this paper explains how. It is important that the proofs produced by automated theorem provers are both understandable and machine checkable. Producing something that satisfies both of these goals is challenging, especially when dealing with complex steps performed by the solver. The main areas where proof output has been improved for understanding include (i) introduction of new symbols (such as Skolem functions) in preprocessing, (ii) representation of unifiers (for example, in resolution steps), and (iii) presentation of AVATAR proofs. These improvements will be illustrated via a number of examples. For checkable proofs Vampire provides a mode that outputs the proof as a number of individual (TPTP) problems that can be independently checked. This process is explained and illustrated with examples.

1 Introduction

Vampire [5] is an automated theorem prover for first-order logic. It was one of the first provers to produce full proofs (it was producing proofs when the CASC competition introduced them in 2001) and supports the TPTP derivation format [12] as well as other output formats.

The proof output of Vampire has changed over the years, following the needs of users and reacting to changes in the prover. This paper will not document these changes but report on the current state. In most cases the notion of *better proof output* is subjective. We have made no attempts to measure this claim but note that better does not necessarily mean shorter but more *usable*.

The most recent changes to proof output where prompted by two developments. Firstly, the introduction of the AVATAR [13, 9, 8] architecture highlighted the need to be able to represent clause splitting in proofs in a reasonable way. Secondly, the growing use of Vampire in program analysis tasks, where proofs are analysed, highlighted the need to make certain proof steps (such as unification) more explicit. These improvements are an ongoing effort.

Another topic that is covered in this paper is that of *proof checking*. This is an area of growing importance as (i) automated theorem provers and the calculi they use become more complex, and (ii) the application of automated theorem provers becomes more automated. There are many examples of soundness bugs only being highlighted in theorem provers a long time after they were introduced, even in Vampire. Being able to *independently* check proofs is essential for the usability of such systems and, as discussed later, any proof format should support proof checking.

The paper is structured as follows. Section 2 reviews the general structure and contents of proofs produced by Vampire. Section 3 describes improvements made to proof steps introducing fresh symbols. Section 4 presents a proof format that makes unifiers apparent. Section 5 discusses how AVATAR proofs are presented. Section 6 gives a note on proof checking. Section 7 concludes with a brief discussion of future plans.

2 Background on Proofs in Vampire

We review the structure and contents of proofs produced by Vampire.

2.1 What is a Proof?

This isn't a paper about logic, theorem proving or proof theory. Other papers exist that can give reasonable introductions to these topics. We assume the reader is broadly familiar with how theorem provers, and Vampire in particular, works. If not, we refer the reader to [5].

In our setting, a proof is a directed-acyclic-graph (DAG) with a single sink node: the empty clause. Source nodes are things that are taken to be true. These are either axioms from the input problem, theory axioms, or definitions introducing fresh symbols to the proof search. A node is a formula (after preprocessing, a clause) and edges represent inference steps where all edges leading into a node form the premises for that inference.

Let us use a short example (also used in [5]) to demonstrate. Consider the following problem about group theory (if all elements in a group have order 2 then the group is Abelian):

Axioms (of group theory):	$\forall x(1 \cdot x = x)$
	$\forall x(x^{-1} \cdot x = 1)$
	$\forall x \forall y \forall z((x \cdot y) \cdot z = x \cdot (y \cdot z))$
Assumptions:	$\forall x(x \cdot x = 1)$
Conjecture:	$\forall x \forall y(x \cdot y = y \cdot x)$

The proof of this problem is given in Figure 1. We have labelled the four separate parts of the proof. These are not the only kinds of steps that can appear in a Vampire proof (see below) but in general the input formulas will appear first, followed by preprocessing and then application of inference rules belonging to some calculus. As mentioned previously, the last step is the derivation of the empty clause.

Each step in the proof has the following structure

```
126.  mult(X1,X2) = mult(X2,X1)  [superposition 24,90]
```

Number — Formula or Clause — Inference — Premises

and therefore represents a node in the proof DAG where each node is numbered so that it can be identified in the premises of another node. This proof step represents the following inference

$$\frac{\texttt{mult(X0, mult(X0, X1))} = \texttt{X1} \qquad \texttt{mult(X2, mult(X1, X2))} = \texttt{X1}}{\texttt{mult(X1, X2)} = \texttt{mult(X2, X1)}}$$

note that no information about unifiers or selected literals (although these are unit clauses) is given, this issue is discussed further in Section 4.

```
                Refutation found.  Thanks to Tanya!
Input
1.  !  [X0] :  mult(e,X0) = X0 [input]
2.  !  [X0] :  e = mult(inverse(X0),X0) [input]
3.  !  [X0,X1,X2] :  mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
4.  !  [X0] :  e = mult(X0,X0) [input]
5.  !  [X0] :  mult(X0,X1) = mult(X1,X0) [input]
Preprocessing
6.  ~!  [X1] :  !  [X0] :  mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
7.  ~!  [X0] :  !  [X1] :  mult(X0,X1) = mult(X1,X0) [rectify 6]
8.  ~!  [X0,X1] :  mult(X0,X1) = mult(X1,X0) [flattening 7]
9.  ?  [X0,X1] :  mult(X0,X1) != mult(X1,X0) [ennf transformation 8]
10.  mult(sK0,sK1) != mult(sK1,sK0) [skolemisation 9]
11.  mult(e,X0) = X0 [cnf transformation 1]
12.  e = mult(inverse(X0),X0) [cnf transformation 2]
13.  mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
14.  e = mult(X0,X0) [cnf transformation 4]
15.  mult(sK0,sK1) != mult(sK1,sK0) [cnf transformation 10]
New Symbols
16.  ~sP2(mult(sK0,sK1)) [inequality splitting name introduction]
17.  sP2(mult(sK1,sK0)) [inequality splitting 15,16]
Superposition Calculus
18.  mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 13,14]
22.  e = mult(X0,mult(X1,mult(X0,X1))) [superposition 13,14]
20.  mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 13,12]
24.  mult(X0,mult(X0,X1)) = X1 [forward demodulation 18,11]
25.  mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 20,11]
30.  mult(inverse(X2),e) = X2 [superposition 24,12]
74.  mult(inverse(X1),e) = mult(X2,mult(X1,X2)) [superposition 25,22]
90.  mult(X2,mult(X1,X2)) = X1 [forward demodulation 74,30]
126.  mult(X1,X2) = mult(X2,X1) [superposition 24,90]
134.  sP2(mult(sK0,sK1)) [backward demodulation 126,17]
135.  $false [subsumption resolution 134,16]
```

Figure 1: A Vampire proof

2.2 Proof Output Formats and Typesetting in LaTeX

Vampire produces various kinds of proof. If one were to run `vampire -explain proof` then Vampire 4.1 would print the explanation:

```
--proof (-p)
        Specifies whether proof will be output. 'proofcheck' will output proof
        as a sequence of TPTP problems to allow for proof-checking.
        default: on
        values: off,on,proofcheck,tptp,smtcomp
```

The off value will suppress any proof output and on will produce proofs in the Vampire-specific proof format shown above (Figure 1). We find this proof format to be the most concise. The proofcheck value will be discussed further in Section 6. The last two values tptp and smtcomp produce proof output in these standardised formats. The TPTP format [12] was designed to be machine parsable and compatible with the TPTP problem language. As SMTCOMP doesn't require or support proofs yet, this proof output just prints sat, unsat or unknown as supported by that competition.

More on TPTP proofs. This format[1] [12] requires each proof step to be recorded as an annotated formula of the form

```
language(name,role,formula,source,useful info).
```

Vampire will use `fof` for the language if no theories are present and `tff` otherwise[2]. If the proof uses the `tff` language then it will also include any necessary type declarations, ensuring that the proof is itself a valid TPTP problem file. Vampire uses the suggested roles of `axiom`, `conjecture`, and `negated_conjecture` for those formulas and `plain` for inferred formulas. The source part is one of three kinds:

- `file` sources describe the file the input formula comes from (and the axiom name in that file if the `--output_axiom_names` option is `on`).
- `inference` source describes an inference. Typically this will include the name of the inference rule and the names of the premises, similar to the Vampire format.
- `introduced` source describes introduced symbols[3], listing the symbols introduced in the proof step. Vampire also uses this source for theory and equality axioms that do not introduce new symbols but are not input axioms.

The first and last source parts indicate proof steps without parents. Vampire doesn't currently use the `useful info` part of annotated formulas but may do so in the future to capture information such as unification positions. Figure 8 (introduced later in Section 5) gives an example of a proof in the TPTP format.

LaTeX output. As well as producing textual proofs, Vampire can also produce proofs in LaTeXby using the `--latex_output` option. This will output a LaTeX file into the given file to produce a proof. For example, Figure 2 gives a proof of Peirce's law produced by Vampire in this way. All LaTeX proof steps given in this paper (and any paper we write using Vampire) are produced in this way. Other modes can also produce LaTeX output. For example, `vampire --mode clausify --latex_output clauses.tex problem.p` will print a LaTeX representation of the clausified problem into `clauses.tex`.

2.3 Possible Proof Steps

For completeness, we provide an (incomplete!) list of steps that can appear in a Vampire proof.

- Inputs:
  - input: an input formula
  - negated conjecture: the conjecture in negated form
  - equality/theory axiom: assumed axioms not in the input
- Preprocessing:
  - flattening, rectify, *nf transformation, skolemisation: clausification steps (see [10])
  - FOOL *: elimination of FOOL elements (see [3, 4])
  - equality proxy *: used to axiomatise equality (necessary for InstGen)
  - definition unfolding, unused predicate definition removal, pure predicate removal: eliminating function and predicate definitions and pure predicates

[1] See http://www.cs.miami.edu/~tptp/TPTP/QuickGuide/Derivations.html for a quick quide.
[2] See [6] for a discussion of how Vampire deals with theories in general.
[3] See http://www.cs.miami.edu/~tptp/TSTP/NewSymbolNames.html for this TPTP convention.

$[7, 6 \to 8$, resolution$]$

$$\frac{p \quad \neg p}{\square}$$

$[4 \to 6$, cnf transformation$]$

$$\frac{(p \vee p) \wedge \neg p}{\neg p}$$

$[5 \to 7$, duplicate literal removal$]$

$$\frac{p \vee p}{p}$$

$[4 \to 5$, cnf transformation$]$

$$\frac{(p \vee p) \wedge \neg p}{p \vee p}$$

$[3 \to 4$, ennf transformation$]$

$$\frac{\neg((\neg p \to p) \to p)}{(p \vee p) \wedge \neg p}$$

$[2 \to 3$, pure predicate removal$]$

$$\frac{\neg(((p \to q) \to p) \to p)}{\neg((\neg p \to p) \to p)}$$

$[1 \to 2$, negated conjecture$]$

$$\frac{((p \to q) \to p) \to p}{\neg(((p \to q) \to p) \to p)}$$

$[1$, input$]$

$$((p \to q) \to p) \to p$$

Figure 2: The LaTeX version of a Vampire proof of Peirce's Law.

  - predicate definition introduction, definition folding: naming subformulas (see Section 3)
  - general splitting: splitting a clause by naming subclauses (see Section 3)
  - inequality splitting: split a clause based on an inequality
- Simplification:
  - duplicate literal removal, trivial inequality removal, distinct equality removal: literal removal operations
  - evaluation: evaluating ground theory terms
- Superposition and Resolution Calculus:
  - resolution
  - subsumption resolution
  - superposition
  - equality factoring
  - equality resolution

```
1. ? [X0] : ! [X1] : ? [X2] : p(X0,X1,X2) [input]
2. ? [X0,X1] : p(X0,a,X1) [input]
3. ~? [X0,X1] : p(X0,a,X1) [negated conjecture 2]
4. ! [X0,X1] : ~p(X0,a,X1) [ennf transformation 3]
5. ? [X0] : ! [X1] : ? [X2] : p(X0,X1,X2) => ! [X1] : ? [X2] :
        p(sK0,X1,X2) [choice axiom]
6. ! [X1] : (? [X2] : p(X0,X1,X2) => p(X0,X1,sK1(X1))) [choice axiom]
7. ! [X1] : p(sK0,X1,sK1(X1)) [skolemisation 1,6,5]
8. p(sK0,X1,sK1(X1)) [cnf transformation 7]
9. ~p(X0,a,X1) [cnf transformation 4]
10. $false [resolution 8,9]
```

Figure 3: Example proof using Skolemisation.

  - backward/forward demodulation
  - unit resulting resolution
- Other:
  - avatar *: steps relating to AVATAR, see Sec. 5 (previously splitting *)
  - extensionality resolution: a special inference step (see [2])
  - instance generation: from the InstGen calculus (see [8])
  - global subsumption: a useful simplification step (see [7, 8])

New inference steps may appear and existing steps may be renamed as Vampire is under constant development.

3 Proof Steps that Introduce Symbols

This section focuses on proof steps that introduce new symbols. Recall that the source nodes of a proof DAG are either input axioms, other assumed axioms (such as equality or theory axioms) or *definitions of new symbols*.

Skolemization

Consider the problem consisting of the axiom $\exists x \forall y \exists z(p(x, y, z))$ and conjecture $\exists x, y(p(x, a, y))$. To show that this is a theorem we need to translate the problem into clausal normal form. One necessary step is Skolemization. In this case the axiom is translated into the clause $p(sk_0, y, sk_1(y))$ where sk_0 is a fresh Skolem constant and sk_1 is a fresh Skolem function. To justify the choice of this new constant we use the following axiom, referred to as a choice axiom:

$$(\exists x : \forall y : \exists z : p(x, y, z)) \Rightarrow (\forall y : \exists z : p(sk_0, y, z))$$

This captures the choice that sk_0 is being used to witness the existentially quantified variable x. Note that this is sound as sk_0 is a fresh constant.

The Vampire proof that the problem is a theorem is given in Figure 3. Note how two choice axioms (5 and 6) are introduced as source nodes of the proof and are then used in the single Skolemisation step (7).

```
1. s(X1,X2) | r(X0,X1) | p(X0,X1) [input]
2. ~p(X1,X1) | s(X0,X0) [input]
3. ~s(a,a) [input]
4. ~r(a,a) [input]
5. s(X1,X2) | sP0(X1) [general splitting component introduction]
6. r(X0,X1) | p(X0,X1) | ~sP0(X1) [general splitting 1,5]
7. s(X0,X0) | sP1 [general splitting component introduction]
8. ~p(X1,X1) | ~sP1 [general splitting 2,7]
9. sP1 [resolution 7,3]
10. sP0(a) [resolution 5,3]
11. p(a,a) | ~sP0(a) [resolution 6,4]
12. p(a,a) [subsumption resolution 11,10]
13. ~sP1 [resolution 12,8]
14. $false [subsumption resolution 13,9]
```

Figure 4: Example proof using General Splitting.

General Splitting

This is a preprocessing step that aims to produce shorter and simpler clauses. It is different from the clause-splitting employed by AVATAR (see Section 5) as it only happens once, hence is allowed to be more expensive. The main difference is the way in which clauses are split. In AVATAR clauses are split into components i.e. minimal variable-disjoint subclauses. However, in general splitting subclauses do not need to be variable-disjoint. Instead, a fresh predicate can be introduced to join the two subclauses on shared variables. As usual, we proceed by example.

Consider the following set of clauses

$$p(x,y) \vee r(x,y) \vee s(y,z)$$
$$s(x,x) \vee \neg p(y,y)$$
$$\neg s(a,a)$$
$$\neg r(a,a)$$

Figure 4 gives the Vampire proof of the unsatisfiability of this clause set. The interesting parts are how the first two clauses in the clause set are split. The clause $p(x,y) \vee r(x,y) \vee s(y,z)$ is split into the clauses $s(y,z) \vee sP_0(y)$ and $r(x,y) \vee p(x,y) \vee \neg sP_0(y)$. The clause $s(x,x) \vee \neg p(y,y)$ is split into the clauses $s(x,x) \vee sP_1$ and $\neg p(y,y) \vee \neg sP_1$, in this case there are no shared variables and therefore a propositional symbol can be used for splitting.

As a side-note, the introduced symbols are placed at the end of the symbol ordering, meaning that they will be the last literals to be selected for resolution etc. This is important, otherwise the first steps of any resolution proof would be to undo the splitting. This can be seen in the proof of Figure 4 i.e. resolving on these literals is delayed until they are units.

Subformula Naming

This is a preprocessing step that aims to minimise the number of clauses produced during clausification. See [10] for further discussion of the technique. The general idea is to name subformulas to avoid (exponential) explosion when expanding operators such as equivalence or

```
1. (h & g & f) | (s & r & q & p) [input]
2. ~p & ~f [input]
3. (s & r & q & p) | ~sP0 [predicate definition introduction]
4. (h & g & f) | sP0 [definition folding 1,3]
5. (s & r & q & p) | ~sP0 [nnf transformation 3]
6. ~sP0 | p [cnf transformation 5]
10. sP0 | f [cnf transformation 4]
13. ~f [cnf transformation 2]
14. ~p [cnf transformation 2]
18. f | p [resolution 10,6]
22. p [subsumption resolution 18,13]
23. $false [subsumption resolution 22,14]
```

Figure 5: Example proof using Subformula Naming.

distributing conjunction over disjunction. This second case will be the source of our (simple) example. Consider the two formulas:

$$(p \wedge q \wedge r \wedge s) \vee (f \wedge g \wedge h)$$
$$\neg f \wedge \neg p$$

Without subformula naming this would expand into 14 clauses, with subformula naming it expands into 9 clauses.

Figure 5 gives the Vampire proof of unsatisfiability for these two formulas[4]. The two key proof steps are predicate definition introduction (3) and definition folding (4). The definition for the subformula is first introduced as a source node in the proof DAG, and then it is *folded* into the input clause to produce a simpler formula.

4 Representing Unifiers in Proofs

A common request from users is to make the unifiers used in proof steps more explicit. There are a few reasons for this. Firstly, to make proofs more readable, it can be difficult to understand how the inference steps have been applied without information about unification. Secondly, some automated analysis of proofs need this information and extracting it, whilst possible, can be difficult as it is necessary to find the positions in each clause to apply unification. Finally, this information may help with *proof checking*, a process discussed later in Section 6. The ideas in this section can be triggered using the `proof_extra` option; see the explanation from your version of Vampire to see what is currently supported as this is an experimental feature.

Many proof steps, such as resolution, superposition and their variants, unify some parts of two clauses to apply the inference. For example, take the following superposition inference rule:

$$\frac{\underline{l \simeq r} \vee C_1 \quad \underline{t[s] \simeq t'} \vee C_2}{(t[r] \simeq t' \vee C_1 \vee C_2)\theta}$$

where θ is a most general unifier of l and s; s is not a variable; $r\theta \not\succeq l\theta$; and $t'\theta \not\succeq t[s]\theta$. To apply the rule one must (i) find the location to perform the unification i.e. the subterm s, and

[4]Note that to produce this proof we turn of `updr`, which would simplify the formulas before subformula naming is applied.

(ii) attempt to produce the most general unifier. Consider the application of the rule in the following proof step from the proof given in Figure 1.

$$\frac{e = mult(inverse(x), x) \qquad mult(x, mult(x, y)) = y}{mult(inverse(z), e) = z}$$

Here $l = mult(inverse(z), z)$ (after variable-renaming) and $s = mult(x, y)$, making the unifier $\theta = [x \mapsto inverse(z), y \mapsto z]$. The question is then how to represent this information in the proof? Once choice would be to give the unifier and positions explicitly. However, this seems redundant as the unifier could be computed from the positions. Therefore, one experimental choice in Vampire is to output position information. The above proof step would be printed as:

```
13. e = mult(inverse(X0),X0) (0:6:1) [cnf transformation 2]
23. mult(X0,mult(X0,X1)) = X1 (1:7:1) [forward demodulation 17,12]
29. mult(inverse(X2),e) = X2 (2:6:1)
      [superposition 23,13, 13 into 23, unify on (0).2 in 13 and (0).1.2 in 23]
```

This identifies the positions in the two premises using the standard position naming scheme (i.e. ordered branches in the syntax tree). However, whilst this can be used to reconstruct the unifier it does not improve readability and may not be that useful for automated techniques. An alternative idea is to apply the unifiers to produce instances of the premises and then perform all inferences without unification. For example, consider the same proof step written as follows:

$$\frac{\dfrac{mult(x, mult(x, y)) = y}{mult(inverse(z), mult(inverse(z), z)) = z} \qquad \dfrac{e = mult(inverse(x), x)}{e = mult(inverse(z), z)}}{mult(inverse(z), e) = z}$$

The above unifier is applied to each side; on the right-side this involves renaming the variables, an implicit step in unification that Vampire avoids where possible by (effectively) producing a symmetric unifier for each side. Now the unifier can be extracted by unifying the instantiated clause with the initial clause. The positions at which unification occurred can be seen from the places where the two clauses differ.

As a further example of this second approach consider the following (inconsistent) clauses:

$$p(f(x, a), g(h(y, x), b)) \vee a \neq f(x)$$
$$\neg p(f(g(h(a, a)), x), y)$$
$$k(x) \neq k(h(a, a)) \vee f(g(x)) = a$$

Figure 6 gives a Vampire proof of this inconsistency where the above technique has been applied. This involves three instantiations and in each case the inferences (`equality resolution`, `superposition` and `subsumption resolution`) are performed without unification. In some cases, for example `equality resolution` (5) the inference step could be replaced by a simpler inference (`trivial inequality removal`) but the inference rule is preserved to demonstrate the reasoning.

We note that unifiers and positions are not the whole story. Inference rules, such as the superposition rule above, also rely on literal selection and term orderings. As you may have noticed from above, the proofs provided by the `proof_extra` options include an extra tuple, e.g. `(2:6:1)`, which gives the `(age:weight:selected)` where `selected` indicates the number of selected literals, *which always appear at the front of the clause* (see [10] for further discussion of literal selection). Vampire doesn't currently output information on term ordering, but currently this should be inferable from the options defining it and the other information available.

```
1. a != f(X0) | p(f(X0,a),g(h(X1,X0),b)) (0:13:1) [input]
2. ~p(f(g(h(a,a)),X0),X1) (0:8:1) [input]
3. k(X0) != k(h(a,a)) | a = f(g(X0)) (0:12:1) [input]
4. k(h(a,a)) != k(h(a,a)) | a = f(g(h(a,a))) (0:16) [instantiation 3]
5. a = f(g(h(a,a))) (1:7:1) [equality resolution 4]
6. a != f(g(h(a,a))) | p(f(g(h(a,a)),a),g(h(X0,
                         g(h(a,a))),b)) (0:22) [instantiation 1]
8. a != a | p(f(g(h(a,a)),a),g(h(X0,g(h(a,a))),b)) (2:18) [superposition 6,5]
9. p(f(g(h(a,a)),a),g(h(X0,g(h(a,a))),b)) (2:15) [trivial inequality removal 8]
11. ~p(f(g(h(a,a)),a),g(h(X0,g(h(a,a))),b)) (0:15) [instantiation 2]
12. $false (2:0) [subsumption resolution 9,11]
```

Figure 6: An example proof where unifiers are shown via instantiation.

5 AVATAR Proofs

AVATAR is a framework for clause splitting that uses a SAT solver to make splitting decisions. This involves introducing propositional names for subclauses and labelling clauses with these names to represent their splitting context. As a further complication, the refutation typically comes from the SAT solver. Previously the way that AVATAR appeared in proofs was quite opaque. This section presents an improved method for writing proofs that make use of AVATAR.

For a full overview of AVATAR we refer the reader to previous publications [13, 9, 8]; indeed, this section may be difficult to follow without some understanding of AVATAR. In this section we consider the worked example from [9]. This example uses the inconsistent clauses

$$q(b) \quad p(x) \vee r(x,z) \quad \neg q(x) \vee \neg s(x) \quad \neg p(x) \vee \neg q(y) \quad s(z) \vee \neg r(x,z) \vee \neg q(w)$$

The Vampire proof of inconsistency is given in Figure 7. In this proof there is a new kind of symbol used in the form of integers, let us call these *assumptions*. Clauses may have assumptions capturing the splitting context they are applicable to. For example, if the unit clause `s(X0)` is dependent on the assumptions `1` and `4` this would be written as `s(X0) <- {0, 4}`. There are five inference rules related to AVATAR and these new assumptions:

- `avatar definition`. This rule introduces definitions for assumptions. As assumptions name components the definition captures this naming. The rule introduces the assumption as a fresh symbol and is source node in the proof DAG.
- `avatar component clause`. This rule introduces the clause derived from the definition that is used in proof search. This indicates that the component is dependent on the assumption that names it.
- `avatar split clause`. This rule uses the definitions to rewrite an existing clause. The rewritten clause will always appear first in the list of premises. This split clause is the propositional clause passed to the SAT solver to make splitting decisions i.e. at least one component needs to be selected in a splitting context.
- `avatar contradiction clause`. This rule uses the definitions to rewrite a conditional contradiction into a propositional clause for the SAT solver. In AVATAR proofs it is common to derive `$false` conditionally many time as each such conditional contradiction represents switching splitting context. However, not all such conditional contradictions will be relevant to the proof.

```
1. q(b) [input]
2. r(X0,X1) | p(X0) [input]
3. ~s(X0) | ~q(X0) [input]
4. ~q(X2) | ~p(X0) [input]
5. ~q(X3) | ~r(X0,X1) | s(X1) [input]
7. 0 <=> ! [X0] : ~p(X0) [avatar definition]
8. ~p(X0) <- {0} [avatar component clause 7]
10. 2 <=> ! [X2] : ~q(X2) [avatar definition]
11. ~q(X2) <- {2} [avatar component clause 10]
12. 0 | 2 [avatar split clause 4,10,7]
14. 4 <=> ! [X1,X0] : (~r(X0,X1) | s(X1)) [avatar definition]
15. ~r(X0,X1) | s(X1) <- {4} [avatar component clause 14]
16. 4 | 2 [avatar split clause 5,10,14]
17. $false <- {2} [resolution 11,1]
18. ~2 [avatar contradiction clause 17,10]
19. s(X0) | p(X1) <- {4} [resolution 15,2]
20. s(X0) <- {0, 4} [subsumption resolution 19,8]
21. ~q(X0) <- {0, 4} [resolution 20,3]
22. 2 | ~0 | ~4 [avatar split clause 21,14,7,10]
23. $false [avatar sat refutation 12,16,18,22]
```

Figure 7: Example proof using AVATAR.

- `avatar sat refutation`. This rule derives a contradiction from the propositional clauses in the SAT solver i.e. the `avatar split clause` and `avatar contradiction clause` bits. The rest of the proof has been about showing how these clauses were derived to end up in the SAT solver to produce a contradiction. Note that a SAT contradiction will typically not involve all SAT clauses passed to the SAT solver. Previously versions of Vampire sometimes reported all such clauses, leading to very large AVATAR proofs.

Hopefully this explanation, perhaps coupled with the worked example from [9], is enough to understand the proof in Figure 7. If not, the `show_avatar` option prints information relevant to AVATAR during proof search (but will slow down proof search).

One benefit of this presentation is that the relevant contents of the SAT solver is immediately evident. If we take the relevant inference rules:

```
12. 0 | 2 [avatar split clause 4,10,7]
16. 4 | 2 [avatar split clause 5,10,14]
18. ~2 [avatar contradiction clause 17,10]
22. 2 | ~0 | ~4 [avatar split clause 21,14,7,10]
```

and strip the unnecessary parts then perform some small formatting operations[5] we get the following problem in the DIMACS format [1] that can be passed to a SAT solver:

```
p cnf 4 4
1 3 0
5 3 0
```

[5]Note that we shifted the variables up one as the DIMACS format reserves 0. This may also need to be normalised so that there are no gaps in the numbering, depending on the SAT solver.

```
-3 0
3  -1 -5 0
```

We note that the TPTP proofs using AVATAR are slightly different as we have to stay within the TPTP requirements. Figure 8 gives the proof of Figure 7 in the TPTP format. Here assumptions are turned into fresh propositional symbols and treated accordingly.

6 A Note on Proof Checking

Proof checking is very important, as discussed in the introduction. If we cannot check that the proof we have produced is correct then we have little confidence in its correctness beyond agreement from other solvers, which may not exist.

To help with proof checking, Vampire can translate its proofs into a series of TPTP problems that can be independently checked. A problem is generated per proof step having the premises as axioms and the conclusion as a conjecture. For example, running the following command (with Vampire 4.1) produces the output given in Figure 9.

```
vampire --decode dis+1002_4_inst=on:tha=off:thf=on_1
        Problems/MSC/MSC023=2.p -p proofcheck
```

This output separates problems for each proof step using the `%#` comment to allow for automatic splitting of the problems. Each problem includes the necessary symbol declarations in the `tff` logic.

The author has a script that passes each problem to a configurable set of independent solvers. This is used during the development of Vampire to check soundness (and can be obtained from the author on request). It, however, has some shortcomings as it cannot handle all preprocessing steps or SAT-based steps such as AVATAR refutation and Global Subsumption.

The set of problems given in Figure 9 have been passed to a set of independent solvers and verified. During our testing it is unusual for a proof step to be unverifiable by any other solver, although it is interesting to note that these proof steps often present difficult problems in themselves e.g. sometimes only one other solver can find a proof and sometimes requiring longer than the original Vampire proof.

The TPTP framework includes a similar tool[6] [11] that the author has not used and cannot comment on the structure of.

7 Conclusion

Vampire has produced readable and usable proofs for a long time. This report discusses the current state of affairs and claims that they are 'better' than they were at some point in the past. The improvements introduced for this paper are an updated presentation of AVATAR proofs and new ideas about how unifiers can be displayed in proofs.

Vampire is under constant development and future work in the area of proof output includes:

- Fully incorporating the ideas of Section 4. Currently these are hidden behind a `proof_extra` option and their implementation has not been optimised.
- Producing proofs in a SMT-LIB compatible format, particularly for proof checking.

[6] See http://www.cs.miami.edu/~tptp/ServiceTools.tgz

```
fof(f1,axiom,(
  q(b)),
  file('MyProblems/avatar_proof.p',unknown)).
fof(f2,axiom,(
  ( ! [X0:$i,X1:$i] : (r(X0,X1) | p(X0)) )),
  file('MyProblems/avatar_proof.p',unknown)).
fof(f3,axiom,(
  ( ! [X0:$i] : (~s(X0) | ~q(X0)) )),
  file('MyProblems/avatar_proof.p',unknown)).
fof(f4,axiom,(
  ( ! [X2:$i,X0:$i] : (~q(X2) | ~p(X0)) )),
  file('MyProblems/avatar_proof.p',unknown)).
fof(f5,axiom,(
  ( ! [X0:$i,X3:$i,X1:$i] : (~q(X3) | ~r(X0,X1) | s(X1)) )),
  file('MyProblems/avatar_proof.p',unknown)).
fof(f7,plain,(
  spl0_0 <=> ! [X0] : ~p(X0)),
  introduced(avatar_definition,[new_symbols(naming,[spl0_0])])).
fof(f8,plain,(
  ( ! [X0:$i] : (~p(X0)) ) | ~spl0_0),
  inference(avatar_component_clause,[],[f7])).
fof(f10,plain,(
  spl0_2 <=> ! [X2] : ~q(X2)),
  introduced(avatar_definition,[new_symbols(naming,[spl0_2])])).
fof(f11,plain,(
  ( ! [X2:$i] : (~q(X2)) ) | ~spl0_2),
  inference(avatar_component_clause,[],[f10])).
fof(f12,plain,(
  spl0_0 | spl0_2),
  inference(avatar_split_clause,[],[f4,f10,f7])).
fof(f14,plain,(
  spl0_4 <=> ! [X1,X0] : (~r(X0,X1) | s(X1))),
  introduced(avatar_definition,[new_symbols(naming,[spl0_4])])).
fof(f15,plain,(
  ( ! [X0:$i,X1:$i] : (~r(X0,X1) | s(X1)) ) | ~spl0_4),
  inference(avatar_component_clause,[],[f14])).
fof(f16,plain,(
  spl0_4 | spl0_2),
  inference(avatar_split_clause,[],[f5,f10,f14])).
fof(f17,plain,(
  $false | ~spl0_2),
  inference(resolution,[],[f11,f1])).
fof(f18,plain,(
  ~spl0_2),
  inference(avatar_contradiction_clause,[],[f17,f10])).
fof(f19,plain,(
  ( ! [X0:$i,X1:$i] : (s(X0) | p(X1)) ) | ~spl0_4),
  inference(resolution,[],[f15,f2])).
fof(f20,plain,(
  ( ! [X0:$i] : (s(X0)) ) | (~spl0_0 | ~spl0_4)),
  inference(subsumption_resolution,[],[f19,f8])).
fof(f21,plain,(
  ( ! [X0:$i] : (~q(X0)) ) | (~spl0_0 | ~spl0_4)),
  inference(resolution,[],[f20,f3])).
fof(f22,plain,(
  spl0_2 | ~spl0_0 | ~spl0_4),
  inference(avatar_split_clause,[],[f21,f14,f7,f10])).
fof(f23,plain,(
  $false),
  inference(avatar_sat_refutation,[],[f12,f16,f18,f22])).
```

Figure 8: Example proof using AVATAR in TPTP format.

```
tff(pred_def_1, type, celsius_fahrenheit_temperature_conversion: ($real * $real) > $o).
tff(r8,conjecture, ( ! [X0:$real] : (celsius_fahrenheit_temperature_conversion(X0,
       $sum($product(1.8,X0),32.0))) ) ). %equality resolution
tff(pr7,axiom, ( ! [X0:$real,X1:$real] : (celsius_fahrenheit_temperature_conversion(X0,
       X1) | $sum($product(1.8,X0),32.0) != X1) ) ).
%#
tff(pred_def_1, type, celsius_fahrenheit_temperature_conversion: ($real * $real) > $o).
tff(r9,conjecture, ( ! [X0:$real,X1:$real] : (~celsius_fahrenheit_temperature_conversion(X0,
       X1) | 451.0 != X1) ) ). %theory flattening
tff(pr6,axiom, ( ! [X0:$real] : (~celsius_fahrenheit_temperature_conversion(X0,451.0)) ) ).
%#
tff(pred_def_1, type, celsius_fahrenheit_temperature_conversion: ($real * $real) > $o).
tff(r10,conjecture, ( ! [X0:$real,X1:$real] : ($sum($product(1.8,X0),32.0) != X1 |
        celsius_fahrenheit_temperature_conversion(X0,X1)) ) ). %theory flattening
tff(pr8,axiom, ( ! [X0:$real] : (celsius_fahrenheit_temperature_conversion(X0,
       $sum($product(1.8,X0),32.0))) ) ).
%#
tff(pred_def_1, type, celsius_fahrenheit_temperature_conversion: ($real * $real) > $o).
tff(r23,conjecture, ( ! [X0:$real] : (celsius_fahrenheit_temperature_conversion(X0,
       $sum($product(1.8,X0),32.0))) ) ). %equality resolution
tff(pr10,axiom, ( ! [X0:$real,X1:$real] : ($sum($product(1.8,X0),32.0) != X1 |
        celsius_fahrenheit_temperature_conversion(X0,X1)) ) ).
%#
tff(pred_def_1, type, celsius_fahrenheit_temperature_conversion: ($real * $real) > $o).
tff(r41,conjecture, ( ! [X0:$real] : ($sum($product(1.8,X0),32.0) != 451.0) ) ). %resolution
tff(pr23,axiom, ( ! [X0:$real] : (celsius_fahrenheit_temperature_conversion(X0,
       $sum($product(1.8,X0),32.0))) ) ).
tff(pr9,axiom, ( ! [X0:$real,X1:$real] : (~celsius_fahrenheit_temperature_conversion(X0,
       X1) | 451.0 != X1) ) ).
%#
tff(pred_def_1, type, celsius_fahrenheit_temperature_conversion: ($real * $real) > $o).
tff(r51,conjecture, ( ! [X0:$real] : (quotient($sum(451.0,-32.0),1.8) != X0) ) ). %evaluation
tff(pr41,axiom, ( ! [X0:$real] : ($sum($product(1.8,X0),32.0) != 451.0) ) ).
%#
tff(pred_def_1, type, celsius_fahrenheit_temperature_conversion: ($real * $real) > $o).
tff(r52,conjecture, ( ! [X0:$real] : (quotient(419.0,1.8) != X0) ) ). %evaluation
tff(pr51,axiom, ( ! [X0:$real] : (quotient($sum(451.0,-32.0),1.8) != X0) ) ).
%#
tff(pred_def_1, type, celsius_fahrenheit_temperature_conversion: ($real * $real) > $o).
tff(r53,conjecture, ( ! [X0:$real] : (232.778 != X0) ) ). %evaluation
tff(pr52,axiom, ( ! [X0:$real] : (quotient(419.0,1.8) != X0) ) ).
%#
tff(pred_def_1, type, celsius_fahrenheit_temperature_conversion: ($real * $real) > $o).
tff(r55,conjecture, 232.778 != 232.778 ). %Instantiation
tff(pr53,axiom, ( ! [X0:$real] : (232.778 != X0) ) ).
%#
tff(pred_def_1, type, celsius_fahrenheit_temperature_conversion: ($real * $real) > $o).
tff(r59,conjecture, $false ). %trivial inequality removal
tff(pr55,axiom, 232.778 != 232.778 ).
%#
```

Figure 9: Example `proofcheck` output for the problem `MSC023=2.p`.

- Extending proof-checking to cover all preprocessing steps and produce the necessary DIMACS file for SAT-based refutations (e.g. AVATAR and Global Subsumption).
- Developing an API where Vampire can be called and a proof object is returned (where appropriate) which can be queried for properties such as unifiers and term orderings.

Finally, if the reader has any requests for improvements in proof output or any other features of Vampire then please contact the author. Similarly for bug reports.

References

[1] DIMACS challenge. Satisfiability. Suggested format. http://www.cs.ubc.ca/~{}hoos/SATLIB/Benchmarks/SAT/satformat.ps, 1993. [Online; accessed 1-September-2016].

[2] Ashutosh Gupta, Laura Kovcs, Bernhard Kragl, and Andrei Voronkov. Extensional crisis and proving identity. In Franck Cassez and Jean-Franois Raskin, editors, *Automated Technology for Verification and Analysis*, volume 8837 of *Lecture Notes in Computer Science*, pages 185–200. Springer International Publishing, 2014.

[3] Evgenii Kotelnikov, Laura Kovács, Giles Reger, and Andrei Voronkov. The vampire and the fool. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 37–48. ACM, 2016.

[4] Evgenii Kotelnikov, Laura Kovács, and Andrei Voronkov. A first class boolean sort in first-order theorem proving and TPTP. In *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*, pages 71–86, 2015.

[5] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *CAV 2013*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35, 2013.

[6] Giles Reger, Nikolaj Bjørner, Martin Suda, and Andrei Voronkov. AVATAR Modulo Theories. In C. Benzmüller, G. Sutcliffe, and R. Rojas, editors, *Proceedings of the 2nd Global Conference on Artificial Intelligence*, EPiC Series in Computing, page To appear. EasyChair Publications, 2016.

[7] Giles Reger and Martin Suda. Global subsumption revisisted (briefly). In Laura Kovács and Andrei Voronkov, editors, *Proceedings of the 3rd Vampire Workshop*, EPiC Series in Computing. EasyChair, 2016.

[8] Giles Reger and Martin Suda. The uses of sat solvers in vampire. In Laura Kovács and Andrei Voronkov, editors, *Proceedings of the 1st and 2nd Vampire Workshops*, volume 38 of *EPiC Series in Computing*, pages 63–69. EasyChair, 2016.

[9] Giles Reger, Martin Suda, and Andrei Voronkov. Playing with avatar. In P. Amy Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 399–415. Springer International Publishing, 2015.

[10] Giles Reger, Martin Suda, and Andrei Voronkov. New Techniques in Clausal Form Generation. In C. Benzmüller, G. Sutcliffe, and R. Rojas, editors, *Proceedings of the 2nd Global Conference on Artificial Intelligence*, EPiC Series in Computing, page To appear. EasyChair Publications, 2016.

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

[12] 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.

[13] Andrei Voronkov. AVATAR: The architecture for first-order theorem provers. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 696–710. Springer International Publishing, 2014.