Some Thoughts About FOL-Translations in Vampire

Giles Reger

University of Manchester, Manchester, U.K.
giles.reger@manchester.ac.uk

Abstract

It is a common approach when faced with a reasoning problem to translate that problem into first-order logic and utilise a first-order automated theorem prover (ATP). One of the reasons for this is that first-order ATPs have reached a good level of maturity after decades of development. However, not all translations are equal and in many cases the same problem can be translated in ways that either help or hinder the ATP. This paper looks at this activity from the perspective of a first-order ATP (mostly Vampire).

1 Introduction

This paper looks at the common activity of encoding problems in first-order logic and running an automated theorem prover (ATP) on the resulting formulas from the perspective of the ATP. This paper focusses on the Vampire [30] theorem prover (available at `https://vprover.github.io/`) but much of the discussion applies to other similarly constructed theorem provers (e.g. E [46] and SPASS [51]).

Over the last few years we have been looking at the application of program analysis/verification, the related encodings, and how Vampire can work with these encodings. In this paper I also mention another setting where we have begun to do some work: working with logics more expressive than first-order logic. This paper comes at the start of an activity to inspect the first-order problems coming from translations involving these logics and considering how we can make Vampire perform better on them.

Throughout the paper I use the terms encoding and translation reasonably interchangeably and lazily. Sometimes the term translation makes more sense (we are moving from one formally defined language to another) and sometimes encoding makes more sense (we are taking a description of a problem and representing it in first-order logic).

This issue of different encodings have differing impacts on how easily a problem is solved is well-known in the automated reasoning community. A standard example is the explosion in conversion to CNF in SAT when not using the Tseitin encoding. The result is obviously bad because it is much larger. In the setting of first-order theorem proving large inputs are also bad but there are other (more subtle) ways in which an encoding can impact the effectiveness of proof search.

The main points of this paper are as follows:

1. The way in which a problem is expressed can have a significant impact on how easy or hard it is for an ATP to solve it and the causes of this go beyond the size of the translation
2. It is often non-obvious whether a translation will be good; we need experiments
3. Sometimes there is no best solution i.e. different encodings may be helpful in different scenarios. In such cases it can be advantageous to move this choice within the ATP

4. Sometimes the only solution is to extend the ATP with additional rules or heuristics to make the ATP treat an encoding in the way we want it to be treated

This points are expanded (with examples) in the rest of the paper.

The rest of the paper is structured as follows. Section 2 describes the relevant inner workings of Vampire that might have a significant impact on the way that it handles different encodings. Section 3 reviews some encodings described in the literature. Section 4 attempts to make some hints about things to consider when designing a representation of a problem in first-order in logic. Section 5 gives some advice on how different encodings should be compared. Section 6 concludes.

2 The Relevant Anatomy of a First-Order ATP

In this section we review the main components of Vampire [30] that might affect how well it handles different encodings. As mentioned above, Vampire shares some elements with other well-known first-order theorem provers.

We consider multi-sorted first-order logic with equality as input to the tool. It will become clear later that Vampire accepts extensions of this in its input, however its underlying logic remains multi-sorted first-order logic with equality and extensions are (mostly) handled as translations into this.

Usually, Vampire deals with the separate notions of *axioms* and *conjecture* (or *goal*). Intuitively, axioms formalise the domain of interest and the conjecture is the claim that should logically follow from the axioms. The conjecture is, therefore, negated before we start the search for a contradiction. Conjectures are captured by the TPTP input language but not by the SMT-LIB input language where everything is an axiom.

2.1 Preprocessing

Vampire works with clauses. So before proof search it is necessary to transform input formulas into this clausal form. In addition to this process there are a number of (satisfiability, but not necessarily equivalence, preserving) optimisation steps. For more information about preprocessing see [17, 43].

Clausification. This involves the replacement of existentially quantified variables by Skolem functions, the expansion of equivalences, rewriting to negation normal form and then application of associativity rules to reach conjunctive normal form. It is well know that this process can lead to an explosion in the number of clauses.

Subformula naming. A standard approach to dealing with the above explosion is the *naming* of subformulas (similar to the Tseitin encoding from SAT). This process is parametrised by a threshold which controls at which point we choose to name a subformula. There is a trade-off here between a (possible) reduction in the size and number of resulting clauses and restricting the size of the signature. Later we will learn that large clauses and a large signature are both detrimental for proof search.

Definition inlining. Vampire detects definitions at the formula and term level. The equivalence $p(X) \leftrightarrow F[X]$ is a definition if p does not occur in formula F, similarly the unit equality

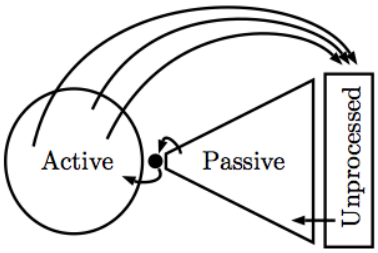

Figure 1: Illustrating the Given Clause Algorithm.

$f(X) = t$ is a definition if f does not appear in term t. Definitions may be *inlined* i.e. all occurrences of p or f are replaced by their definition. This reduces the size of the signature with the cost of a potential increase in the size of clauses. Note that it can be quite easy to break this notion of definition in encodings. For example, by introducing a guard to a definition.

Goal-based premise selection. Vampire can use a goal to make proof search more goal-directed. The point here is that if an encoding does not explicitly highlight the goal we cannot make use of these techniques. There are two techniques used in preprocessing. The first is SInE selection [19] which *heuristically* selects a subset of axioms that are likely to be used in the proof of the goal. Axioms are selected using a notion of closeness to the goal that is based on whether they can be connected to the goal via their least common symbol. The second is the set of support strategy [52] where clauses from the goal are put into a set of support and proof search is restricted so that it only makes inferences with clauses in, or derived from, this set.

2.2 Saturation-Based Proof Search

After a clause set has been produced, Vampire attempts to *saturate* this set with respect to some inference system $\mathcal{I}$. The clause set is saturated if for every inference from $\mathcal{I}$ with premises in S the conclusion of the inference is also added to S. If the saturated set S contains a contradiction then the initial formulas are unsatisfiable. Otherwise, if $\mathcal{I}$ is a complete inference system and, importantly, the requirements for this completeness have been preserved, then the initial formulas are satisfiable. Finite saturation may not be possible and many heuristics are employed to make finding a contradiction more likely.

To compute this saturation we use a set of active clauses, with the invariant that all inferences between active clauses have been performed, and a set of passive clauses waiting to be activated. The algorithm then iteratively selects a *given clause* from passive and performs all necessary inferences to add it to active. The results of these inferences are added to passive (after undergoing some processing). This is illustrated in Figure 1. An important aspect of this process is *clause selection*. Clauses are selected either based on their age (youngest first) or their weight (lightest first) with these two properties being alternated in some specified ratio.

A recent addition to this story is AVATAR [50, 40], which (optionally) performs *clause splitting* using a SAT solver. The main point here is that the success of AVATAR is driven by the observation that saturation-based proof search does not perform well with long or heavy clauses. Therefore, encodings should avoid the introduction of such clauses. As an additional point, AVATAR can only be utilised if the boolean structure of a problem is exposed at the literal-level. For example, including a predicate *implies* with associated axioms would not play

to AVATAR's strengths.

2.3 Inference Rules

Vampire uses resolution and superposition as its inference system $\mathcal{I}$ [1, 34]. A key feature of this calculus is the use of *literal selection* and *orderings* to restrict the application of inference rules, thus restricting the growth of the clause sets. Vampire uses a Knuth-Bendix term ordering (KBO) [23, 25, 32] which orders terms first by weight and then by symbol precedence whilst agreeing with a multisubset ordering on free variables. The symbol ordering is taken as a parameter but is relatively coarse in Vampire e.g. by order of occurrence in the input, arity, frequency or the reverse of these. There has been some work beginning to explore more clever things to do here [22, 38] but we have not considered treating symbols introduced by translations differently (although they will appear last in occurrence).

Understanding this is important as some translations change symbols used in terms, which can change where the term comes in the term ordering. Vampire makes use of both complete and incomplete literal selection functions [18] which combine the notion of maximality in the term ordering with heuristics such as selecting the literal with the least top-level variables.

Another very important concept related to saturation is the notion of *redundancy*. The idea is that some clauses in S are *redundant* in the sense that they can be safely removed from S without compromising completeness. The notion of saturation then becomes saturation-up-to-redundancy [1, 34]. An important redundancy check is *subsumption*. A clause A subsumes B if some subclause of B is an instance of A, in which case B can be safely removed from the search space as doing so does not change the possible models of the search space S. The fact that Vampire removes redundant formulas is good but if there is a lot of redundancy in the encoding we can still have issues as this removal can be lazy (e.g. when using the discount saturation loop that does not remove redundancies from the passive set).

Figure 2 gives a selection of inference rules used in Vampire. An interesting point can be illustrated by examining the demodulation rule. This uses unit equalities to rewrite clauses to replace larger terms by smaller terms. A similar, but more general, rewriting is performed by superposition. Ordering restrictions in inference rules make proof search practical but, as I comment later, can mean that the theorem prover does not treat encoded rules in the way that we want.

2.4 Strategies and Portfolio Mode

Vampire is a portfolio solver [36]. It implements many different techniques and when solving a problem it may use tens to hundreds of different strategies in a time-sliced fashion. Along with the above saturation-based proof search method, Vampire also implements the InstGen calculus [24] and a finite-model building through reduction to SAT [42]. In addition, Vampire can be run in a parallel mode where strategies are distributed over a given number of cores.

This is important as Vampire can try different preprocessing and proof search parameters in different strategies, meaning that it does not matter if a particular encoding does not work well with one particular strategy. Furthermore, if Vampire is allowed to do the translation itself then it can try multiple different translations during proof search.

2.5 Problem Characteristics that Matter

As summary, we can discuss the problem characteristics that matter. The main ones we usually talk about are:

Resolution **Factoring**

$$\frac{\underline{A} \vee C_1 \quad \underline{\neg A'} \vee C_2}{(C_1 \vee C_2)\theta}\ , \qquad \frac{\underline{A} \vee A' \vee C}{(A \vee C)\theta}\ ,$$

where, for both inferences, $\theta = \mathsf{mgu}(A, A')$ *and* A *is not an equality literal*

Superposition

$$\frac{\underline{l \simeq r} \vee C_1 \quad \underline{L[s]_p} \vee C_2}{(L[r]_p \vee C_1 \vee C_2)\theta} \quad \textit{or} \quad \frac{\underline{l \simeq r} \vee C_1 \quad \underline{t[s]_p \otimes t'} \vee C_2}{(t[r]_p \otimes t' \vee C_1 \vee C_2)\theta}\ ,$$

where $\theta = \mathsf{mgu}(l, s)$ *and* $r\theta \not\succeq l\theta$ *and, for the left rule* $L[s]$ *is not an equality literal, and for the right rule* $\otimes$ *stands either for* $\simeq$ *or* $\not\simeq$ *and* $t'\theta \not\succeq t[s]\theta$

EqualityResolution **EqualityFactoring**

$$\frac{\underline{s \not\simeq t} \vee C}{C\theta}\ , \qquad \frac{\underline{s \simeq t} \vee s' \simeq t' \vee C}{(t \not\simeq t' \vee s' \simeq t' \vee C)\theta}\ ,$$

where $\theta = \mathsf{mgu}(s, t)$ *where* $\theta = \mathsf{mgu}(s, s')$, $t\theta \not\succeq s\theta$, *and* $t'\theta \not\succeq s'\theta$

Demodulation **UnitResultingResolution**

$$\frac{l \simeq r \quad \cancel{L[l\theta] \vee C}}{(L[r\theta] \vee C)\theta}\ , \qquad \frac{C \vee A_1 \vee \ldots A_n \quad \neg B_1 \quad \ldots \quad \neg B_n}{C\theta}\ ,$$

where $l\theta \succ r\theta$ *where* $|C| \leq 1$ *and* $\theta = \bigsqcup \mathsf{mgu}(A_i, B_i)$

Figure 2: Selected inference rules.

- Number of resulting clauses. Clearly, if there are more clauses it will take longer to process them initially. However, the number of clauses is a bad proxy for effort as a small clause set can lead to many consequences, where a large clause set may have almost no consequences at all.

- Size of resulting clauses. Long and heavy clauses lead to even longer and heavier clauses when applying rules such as resolution. Some processes, such as subsumption checking, are exponential in the length of a clause.

- Size of signature. In SAT-solving the relationship with the signature is clear as each symbol represents a potential choice point. In saturation-based proof search we are effectively trying to eliminate the literals in clauses until we get an empty clause. The combination of clause selection and literal selection steers this process. The notion of maximality built into literal selection means that 'bigger' symbols are preferred, driving proof search to effectively eliminate symbols in this order. So, like in SAT solving, the larger the signature the more work we need to do.

But as we can see from the discussions of literal selection in this paper, issues with encodings can be more subtle than this.

3 Examples of Translations/Encodings

This section reviews some translations or encodings targeting first-order logic. I don't attempt to be exhaustive; in fact there are many well-established examples that I will have missed.

3.1 Simplifying things further

Not all ATPs can handle multi-sorted first-order logic with equality. There are methods that can be used to remove things that are not supported or wanted.

Removing sorts. It is not always possible to simply drop sort information from problems as the 'size' of different sorts may have different constraints [11]. Two proposed solutions are to either *guard* the use of sorted variables by a *sort predicate* that indicates whether a variable is of that sort (this predicate can be set to false in a model for all constants not of the appropriate sort) or *tag* all values of a sort using a *sort function* for that sort (in a model the function can map all constants to a constant of the given sort). Both solutions add a lot of clutter to the signature although sort predicates add more as they also require the addition of axioms for the sort predicates. Experimental evaluation [8, 42] concluded that the encodings can be complementary. Similar techniques can be used for removing polymorphism [8]. An important optimisation here is that we can simply drop sorts if they are monotonic [11].

Removing equality. Every atom $t = s$ can be replaced by $eq(t, s)$ with the addition of axioms for reflexivity, symmetry, transitivity of eq and congruence of functions and predicates. However, the result is not able to use the efficient superposition or demodulation rules. Vampire optionally makes this transformation as in some cases it can aid proof search, in particular when we do not require deep equality reasoning. It is necessary to remove equality when using InstGen as this does not support equality.

Removing functions. A well known decidable fragment of first-order logic is the Bernays-Schönfinkel fragment (also known as *effectively propositional*) where formulas contain no (non-zero arity) function symbols (even after Skolemisation). If a problem fits in this fragment then we obtain this useful property. For example, the problem of finding a finite model of a first-order formula can be reduced to a sequence of effectively propositional problems [4].

3.2 Syntactic Extensions for Program Verification

A common setting where we see problems encoded in first-order logic is for program verification. Tools such as Boogie [31] and Why3 [10] produce first-order proof obligations. There tend to be common expressions in such obligations, such as *if-then-else*, which is why languages such as TPTP [48, 49] and SMT-LIB [3] support these features. However, ATPs typically do not support these features directly but via the following translations.

Boolean sort. In first-order logic predicates are of boolean sort and functions are not. However, in programs we typically want to reason about boolean functions, which requires a first-class boolean sort in the problem. This can be encoded in first-order logic (as explained in the FOOL work [26, 28]) by the introduction of two constants *true* and *false* and two axioms $\mathit{true} \neq \mathit{false}$ and $\forall x \in \mathit{bool} : x = \mathit{true} \vee x = \mathit{false}$. This second axiom is problematic as it will unify with every boolean sorted term and assert that it is either true or false. To overcome this we introduce a specialised inference rule that captures the desired behaviour without the explosive nature [26].

If-then-else. Conditionals are a common construct in programs and it is important to model them. This is often done by extending the syntax by a term of the form $\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t$ for a boolean sorted term b and two terms of the same sort t and s. The question is then how this should be translated to first-order logic. I discuss three alternatives. In each case we assume we are given a formula containing an if-then-else term e.g. $F[\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t]$. In the first case we translate this formula into two formulas

$$b \rightarrow F[s] \quad , \quad \neg b \rightarrow F[t]$$

where b and its negation are used as guards. This has the disadvantage that it copies F. An altnernative is to produce the three formulas

$$F[g(X)] \quad , \quad b \rightarrow g(X) = s \quad , \quad \neg b \rightarrow g(X) = t$$

where g is a fresh symbol and X are the free variables of the original if-then-else term. This introduces a new symbol g. Finally, we could replace the formula by $F[\mathit{ite}_\tau(b, s, t)]$, where ite is a fixed function symbol of the sort $\mathit{bool} \times \tau \times \tau \rightarrow \tau$, and add the general axioms

$$(X = \mathit{true}) \rightarrow \mathit{ite}_\tau(X, s, t) = s \quad , \quad (X = \mathit{false}) \rightarrow \mathit{ite}_\tau(X, s, t) = t$$

capturing the intended behaviour of ite_τ. This only introduce a pair of axioms per set of if-then-else expressions with the same resultant sort.

We now have three different translations and the question is: which should we use? There are various observations we can make at this point. The first translation copies F, which may lead to many more clauses being introduced if F is complex. However, it does not extend the signature, whilst the second two translations do. The last translation introduces some axioms that are likely to be quite productive during proof search. We should also think about what we want to happen here; ideally the guards will be evaluated first and then the rest of the expression either selected or thrown away. Recall that *literal selection* is the mechanism for choosing which part of a clause is explored next. In general, literal selection prefers heavier literals that are more ground and dislikes positive equalities. Therefore, in the second translation it is likely that the guards will be evaluated before the equalities. In general, it is likely that the guard will be simpler than the conditional body and therefore the first translation is likely to not do what we want. However, it is difficult to draw conclusions from this; as discussed at the end of this paper, we should really run some experiments to see. Note that currently Vampire will pick between the first two depending on the subformula naming threshold [27].

Next state relations. Recent work [12] has used a tuple-based let-in expression to encode next-state relations of loop-free imperative programs. These tuple-based let-in expressions are then translated into clausal form by introducing names for the bound functions [26] and relevant axioms for the tuples (if required). The point of this translation was to avoid translation to

single static assignment form as is often done in translations of imperative programs [2]. The reason to avoid such a form is that it drastically increases the size of the signature. Another benefit of this work is that we can translate loop-free imperative programs directly into first-order logic, allowing Vampire to take such programs directly in its input.

3.3 Proofs about Programming Languages

Recent work [15, 14] has used ATPs to aid in establishing soundness properties of type systems. The idea is to translate a language specification and an 'exploration task' into first-order logic. They define and evaluate 36 different compilation strategies for producing first-order problems.

The most interesting observation to make about this work is that they reinvent a number of encodings and fail to take advantage of the encodings that already exist within ATPs. For example, they consider if-then-else and let-in expressions but do not encode these in the TPTP format but choose their own encodings. In this case they make the first choice for encoding conditionals given above and they choose an alternative way to capture let-in expressions. They also explore the inlining of definitions in their encoding, a step that Vampire already takes in preprocessing. Even though they were able to use more contextual information to decide when to inline, they found that their inlining had minimal impact on performance.

A positive point about this work is that they made a thorough comparison of the different encodings, although some of the suggested encodings were likely to hinder rather than help the ATP. In particular, those that remove information from the problem.

3.4 Higher-Order Logic

Many of the automated methods for reasoning in higher-order logic work via (a series of) encodings into first-order logic. Well-known examples are the Sledgehammer tool [9, 35] and Leo III [47]. The translation of higher-order logic to first-order logic has been well studied [33]. In the translation it is necessary to remove three kinds of higher-order constructs and we discuss the standard translation for removing these here.

Handling partial application and applied variables. In higher-order formulas it is possible to partially apply function symbols, which means that we cannot use the standard first-order term structure. The standard solution is to use the so-called *applicative form*. The general idea is to introduce a special *app* symbol (per pair of sorts in the multi-sorted setting). An application st is then translated as $app(s,t)$. This also addresses the issue that higher-order formulas can contain applied variables e.g. $\exists f.\forall x.f(x) = x$ now becomes $\exists f.\forall x.app(f,x) = x$.

Whilst this translation is correct it also adds a lot of clutter to clauses that causes problems for proof search. One example of this is in literal selection as applicative form can change the notion of maximal literal. For example, the literal $g(a) = a$ will be maximal in clause $f(a) = a \vee g(b) = b$ if $g \succ f$ in the symbol ordering. However, in $app_{\tau_1}(f,a) = a \vee app_{\tau_2}(g,b) = b$ the maximal literal depends on the ordering of app_{τ_1} and app_{τ_2}, which cannot consistently agree with the symbol ordering (consider h of sort τ_1 such that $h \succ g$). As a result, it seems likely that the number of maximal literals in clauses will significantly increase. Additionally, literal selection heuristics often consider the number of 'top' variables of a literal but the applicative form changes the height of variables in terms. Finally, the applicative form can decrease the efficiency of term indexing as (i) these often use function symbols to organise the tree but these have been replaced, and (ii) terms are larger making the indices larger.

However, the applicative form is only needed for supporting partial application and applied variables. If neither are used in a problem then it is not needed (although note that the combinator translation discussed below for λ-expressions introduces applied variables).

Removing λ-expressions. The standard approach to removing λ-expressions is rewriting with Turner combinators [33] and the addition of axioms defining these combinators. The translation itself can lead to very large terms, which can be mitigated by the introduction of additional axioms at the cost of further axioms and extensions to the signature. An alternative is λ-*lifting* which introduces new function symbols for every nested λ-expression, which again can significantly increase the size of the signature. To avoid combinator axioms cluttering proof search we have begun to explore replacing these with (incomplete) inference rules emulating their behaviour [6].

Translating logical operators. It is sometimes necessary to translate logical operators occurring inside λ-expressions (other embedded logical operators can be lifted via naming). These must then be specified by axioms e.g. OR could be defined by

$$app(app(\mathsf{OR}, x), y) = true \iff X = true \ \vee \ Y = true$$

which produces the clauses

$$app(app(\mathsf{OR}, X), Y) = false \ \vee \ X = true \ \vee \ Y = true$$
$$app(app(\mathsf{OR}, X), Y) = true \ \vee \ X = false$$
$$app(app(\mathsf{OR}, X), Y) = true \ \vee \ Y = false$$

Alternatively, the last clause could be dropped and replaced by

$$app(app(\mathsf{OR}, X), Y) = app(app(\mathsf{OR}, Y), X)$$

The non-orientability of this last equality suggests that it would be too productive. Finally, whilst it seems most natural to introduce an equivalence, we would not typically need to reason in the other direction and it could be interesting to see how necessary the first clause is.

3.5 Other Logics

As we have seen from the previous example, a common activity is to consider other logics as (possibly incomplete) embeddings into first-order logic. In all cases, an interesting direction of research will be to inspect the kinds of problems produced and consider whether the encoding or ATP could be improved.

Translations via higher-order logic. Many non-classical logics, included quantified multi-modal logics, intuitionistic logic, conditional logics, hybrid logic, and free logics can be encoded in higher-order logic [5, 13].

Ontology Languages. In this setting it is more typical to use optimised reasoners for decidable logics. However, some work has looked at translating these logics into first-order logic and employing an ATP. As two examples, Schneider and Sutcliffe [45] give a translation of OWL 2 Full into first-order logic and Horrocks and Voronkov [20] translate the KIF language. In both cases the translations allowed ATPs to solve difficult problems but it was not clear whether they were in any sense optimal.

Propositional Modal Logic. There exists a standard relational encoding of certain propositional modal logics into first-order logic where the accessibility relation is given as a predicate R and relational correspondence properties are captured as additional axioms. To combat certain deficiencies in this translation, a functional translation method was introduced with different variations [21]. One observation here is that many of the relational correspondence properties are not friendly for proof search and it may be interesting to explore specific ATP extensions that could help with such translations.

4 Improving a Translation

When thinking about translations of problems to first-order logic there are things that can be done to improve ATP performance within the translation and there are things that can only be done within the ATP. I discuss both sets of improvements here.

4.1 What Can be Controlled in an Encoding

The following describes some things to watch out for when designing a translation.

Throwing things away. A common mistake in encodings is to throw away information that the theorem prover can use in proof search. An obvious example of this is performing Skolemisation and dropping information about which function symbols are Skolem functions – in some applications, e.g. inductive theorem proving, such Skolem constants play a special role in proof search. Another example is dropping information about which formula is the goal. A more subtle example of this is to perform subformula naming before passing the problem to Vampire. In this case the original structure is lost and it is not possible for Vampire to choose to name fewer subformulas. Recall that an advantage of the strategy scheduling structure of Vampire is that it can try out various different preprocessing steps. In general, if an ATP can perform a preprocessing step then it should be given the option to.

Not adding what is helpful. Quite often there is some information about a problem that may not affect the solvability of the problem but can be useful in improving the performance of the solver. An obvious example is in the translation of the problem of finding first-order models to EPR [4]. In this translation there are a number of symmetries that are known during the translation but that would be expensive to recover after the fact. Failing to add these gives the solver unnecessary work to do. Another example would be failing to exclude (sets of) axioms that are known to have no relation to the current goal – something that may be known when generating the problem but can be difficult to determine by the ATP.

4.2 What Needs ATP Support

Here I discuss some issues with translations that require ATP support to handle.

Exploding axioms. It is quite common to include an axiomitisation of some theory that is included in the problem but only needed a little bit in proof search. It is also common for these axioms to be explosive, in the sense that they generate a lot of unnecessary consequences in the search space. A typical example would be axioms for arithmetic, or even for equality. We have already seen an example with this when encoding the boolean sort.

One ATP-based solution to this that we have explored [39, 7] is to identify such axioms and limit the depth to which these axioms can interact. This is effectively the same as precomputing the set of consequences of the axioms up to a certain size but happens dynamically at proof search so may not require the full set. An alternative ATP-based solution is to capture the rules represented by the axioms as additional inference rules.

Rewriting the wrong way. Sometimes rules may not do what we want them to do. For example, the following rule for set extensionality

$$(\forall x)(\forall y)(((\forall e)(e \in y \leftrightarrow e \in y)) \rightarrow x = y)$$

will be clausified as

$$f(x,y) \notin x \vee f(x,y) \notin y \vee x = y$$

which is correct but will not be treated in the way we want by the ATP. Literal selection prefers larger literals and dislikes (positive) equalities. Therefore, the literal $x = y$ will not be selected. However, given the goal $a \neq b$ for sets a and b we want this to unify with $x = y$ to generate the necessary subgoals. The rewrite rule is oriented the wrong way. This can also happen with implications or equalities where we want to rewrite something small into something big.

One ATP-based solution is to introduce specific rules to handle special cases. For example, in the case of extensionality Vampire includes the inference rule [16]

$$\frac{x \simeq y \vee C \quad \underline{s \not\simeq t} \vee D}{C\{x \mapsto s, y \mapsto t\} \vee D},$$

where $x \simeq y \vee C$ is identified as an extensionality clause.

Finding the goal. As mentioned previously, preprocessing and proof search can benefit from knowing what the goal of a problem is. Ideally the translation preserves this but the ATP can also attempt to guess the goal based on the location of axioms in the input problem and the frequency of symbols occurring in different parts of the problem [37].

Changing the translation. It can be possible to detect cases where an alternative encoding may be preferable and switch to that encoding via a further translation step. For example, some work [44] in the CVC4 SMT solver looked at identifying datatypes encoding natural numbers and translating these to guarded usage of integers, which the solver has better support for.

Supporting multiple encodings. Ultimately we may want to try various encodings via the strategy scheduling approach. However, if the ATP cannot understand the original problem then it cannot natively support these encodings. The solution here is to extend the input language of the ATP such that it supports additional features that allow the different encodings to take place. This is what we have done with our work on programs [12] and datatypes [29].

5 Comparing Encodings

It is generally hard to evaluate the addition of a new feature to a first-order ATP [41]. The same can be said for encodings. Here I give a few thoughts about how to go about it:

1. *Actually do it.* Without experiments it is impossible to draw any real conclusions. A corollary here is *don't rely on properties of the output of the encoding that you assume are good proxies for performance* e.g. the number of resulting clauses. Whilst bigger problems can pose a problem for ATPs, there are many small (< 50 clauses) problems that ATPs find very challenging.

2. *Use portfolio mode.* The encoding may require certain preprocessing or proof search parameters to be switched on (or off) to be effective. Running in a single mode may miss this and the wrong conclusion may be drawn. Furthermore, the encoding may react positively to multiple different strategies and similarly, without portfolio mode this will be missed. A corollary here is to look at the strategies actually used to solve problems and see if there are any common patterns.

3. *Don't use portfolio mode.* The portfolio modes in a solver are tuned to the current options and assumptions about input problems. They are usually slightly over-fitted. Perhaps the parameter option needed for your encoding is not included. The obvious solution is to search the parameter space for the optimal combination of parameters for a particular encoding. This is what we often do in Vampire but it is very expensive.

4. *Think about the resources.* Do the input problems reflect what you care about; often it is easy to come up with pathological bad cases but optimising for these often makes little practical difference. Is the time limit sensible; for portfolio mode allow minutes (I use 5) and for single strategies allow seconds (I use 10 to 30) but use cases vary. The point here is whether the results reflect actual usage for the problem being targeted – one encoding may work better than another for cases that you don't care about.

6 Conclusion

In this paper I aimed to give some thoughts about the activity of translating problems into first-order logic. My main conclusion is that for some encodings we need ATP support, and this is what we are trying to provide in Vampire. If you want help extending Vampire for a particular encoding please contact me.

References

[1] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science, 2001.

[2] Michael Barnett and K. Rustan M. Leino. To goto where no statement has gone before. In *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings*, pages 157–168, 2010.

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[4] Peter Baumgartner, Alexander Fuchs, Hans de Nivelle, and Cesare Tinelli. Computing finite models by reduction to function-free clause logic. *Journal of Applied Logic*, 7(1):58 – 74, 2009. Special Issue: Empirically Successful Computerized Reasoning.

[5] Christoph Benzmüller and Lawrence C Paulson. Multimodal and intuitionistic logics in simple type theory. *Logic Journal of IGPL*, 18(6):881–892, 2010.

[6] Ahmed Bhayat and Giles Reger. Higher-order reasoning vampire style. In *25th Automated Reasoning Workshop*, page 19, 2018.

[7] Ahmed Bhayat and Giles Reger. Set of support for higher-order reasoning. In *PAAR 2018*, 2018.

[8] J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Encoding monomorphic and polymorphic types. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013)*, 2013.

[9] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. *J. Autom. Reasoning*, 51(1):109–128, 2013.

[10] Franois Bobot, Jean christophe Fillitre, Claude March, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *In Workshop on Intermediate Veri cation Languages*, 2011.

[11] Koen Claessen, Ann Lilliesträm, and Nicholas Smallbone. Sort it out with monotonicity - translating between many-sorted and unsorted first-order logic. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, pages 207–221, 2011.

[12] Laura Kovcs Evgenii Kotelnikov and Andrei Voronkov. A FOOLish encoding of the next state relations of imperative programs. In *IJCAR 2018*, 2018.

[13] Tobias Gleißner, Alexander Steen, and Christoph Benzmüller. Theorem provers for every normal modal logic. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, pages 14–30, 2017.

[14] Sylvia Grewe, Sebastian Erdweg, and Mira Mezini. Using vampire in soundness proofs of type systems. In *Proceedings of the 1st and 2nd Vampire Workshops, Vampire@VSL 2014, Vienna, Austria, July 23, 2014 / Vampire@CADE 2015, Berlin, Germany, August 2, 2015*, pages 33–51, 2015.

[15] Sylvia Grewe, Sebastian Erdweg, André Pacak, Michael Raulf, and Mira Mezini. Exploration of language specifications by compilation to first-order logic. *Sci. Comput. Program.*, 155:146–172, 2018.

[16] 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.

[17] Krystof Hoder, Zurab Khasidashvili, Konstantin Korovin, and Andrei Voronkov. Preprocessing techniques for first-order clausification. In *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, pages 44–51, 2012.

[18] Kryštof Hoder, Giles Reger, Martin Suda, and Andrei Voronkov. Selecting the selection. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 – July 2, 2016, Proceedings*, pages 313–329, Cham, 2016. Springer International Publishing.

[19] Krystof Hoder and Andrei Voronkov. Sine qua non for large theory reasoning. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, pages 299–314, 2011.

[20] Ian Horrocks and Andrei Voronkov. Reasoning support for expressive ontology languages using a theorem prover. In *Proceedings of the 4th International Conference on Foundations of Information and Knowledge Systems*, FoIKS'06, pages 201–218, Berlin, Heidelberg, 2006. Springer-Verlag.

[21] Ullrich Hustadt and Renate A. Schmidt. MSPASS: Modal reasoning by translation and first-order resolution. In Roy Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 67–71, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[22] Jan Jakubuv, Martin Suda, and Josef Urban. Automated invention of strategies and term orderings for vampire. In *GCAI 2017, 3rd Global Conference on Artificial Intelligence, Miami, FL, USA, 18-22 October 2017.*, pages 121–133, 2017.

[23] D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor,

Computational Problems in Abstract Algebra, pages 263–297. Pergamon Press, Oxford, 1970.

[24] Konstantin Korovin. Inst-Gen – A Modular Approach to Instantiation-Based Automated Reasoning. In Andrei Voronkov and Christoph Weidenbach, editors, *Programming Logics: Essays in Memory of Harald Ganzinger*, pages 239–270, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[25] Konstantin Korovin and Andrei Voronkov. Orienting rewrite rules with the Knuth–Bendix order. *Inf. Comput.*, 183(2):165–186, June 2003.

[26] 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.

[27] Evgenii Kotelnikov, Laura Kovács, Martin Suda, and Andrei Voronkov. A clausal normal form translation for FOOL. In *GCAI 2016. 2nd Global Conference on Artificial Intelligence, September 19 - October 2, 2016, Berlin, Germany*, pages 53–71, 2016.

[28] 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.

[29] Laura Kovács, Simon Robillard, and Andrei Voronkov. Coming to terms with quantified reasoning. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 260–270. ACM, 2017.

[30] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *CAV 2013*, volume 8044 of *LNCS*, pages 1–35, 2013.

[31] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 312–327, Berlin, Heidelberg, 2010. Springer-Verlag.

[32] Michel Ludwig and Uwe Waldmann. An extension of the Knuth-Bendix ordering with LPO-like properties. In *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, pages 348–362, 2007.

[33] Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, Jan 2008.

[34] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.

[35] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *IWIL 2010. The 8th International Workshop on the Implementation of Logics*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2012.

[36] Michael Rawson and Giles Reger. Dynamic strategy priority: Empower the strong and abandon the weak. In *PAAR 2018*, 2018.

[37] Giles Reger and Martin Riener. What is the point of an smt-lib problem? In *16th International Workshop on Satisfiability Modulo Theories*, 2018.

[38] Giles Reger and Martin Suda. Measuring progress to predict success: Can a good proof strategy be evolved? *AITP 2017*, 2017.

[39] Giles Reger and Martin Suda. Set of support for theory reasoning. In *IWIL Workshop and LPAR Short Presentations*, volume 1 of *Kalpa Publications in Computing*, pages 124–134. EasyChair, 2017.

[40] 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, Cham, 2015. Springer International Publishing.

[41] Giles Reger, Martin Suda, and Andrei Voronkov. The challenges of evaluating a new feature 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 70–74. EasyChair, 2016.

[42] Giles Reger, Martin Suda, and Andrei Voronkov. Finding finite models in multi-sorted first-order logic. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016: 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 323–341. Springer International Publishing, 2016.

[43] Giles Reger, Martin Suda, and Andrei Voronkov. New techniques in clausal form generation. In Christoph Benzmüller, Geoff Sutcliffe, and Raul Rojas, editors, *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, volume 41 of *EPiC Series in Computing*, pages 11–23. EasyChair, 2016.

[44] Andrew Reynolds and Viktor Kuncak. Induction for SMT solvers. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, pages 80–98, 2015.

[45] Michael Schneider and Geoff Sutcliffe. Reasoning in the OWL 2 full ontology language using first-order automated theorem proving. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, pages 461–475, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[46] S. Schulz. E — a brainiac theorem prover. 15(2-3):111–126, 2002.

[47] Alexander Steen and Christoph Benzmüller. The higher-order prover Leo-III. *CoRR*, abs/1802.02732, 2018.

[48] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.

[49] Geoff Sutcliffe and Evgenii Kotelnikov. TFX: The TPTP extended typed first-order form. In *PAAR 2018*, 2018.

[50] 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.

[51] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.

[52] Lawrence Wos, George A. Robinson, and Daniel F. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *J. ACM*, 12(4):536–541, October 1965.