Representation, Verification, and Visualization of Tarskian Interpretations for Typed First-order Logic

Alexander Steen\textsuperscript{1}, Geoff Sutcliffe\textsuperscript{2}, Pascal Fontaine\textsuperscript{3}, and Jack McKeown\textsuperscript{2}

\textsuperscript{1} University of Greifswald, Greifswald, Germany  
alexander.steen@uni-greifswald.de

\textsuperscript{2} University of Miami, Miami, USA  
geoff@cs.miami.edu, jam771@miami.edu

\textsuperscript{3} University of Liège, Liège, Belgium  
Pascal.Fontaine@uliege.be

Abstract

This paper describes a new format for representing Tarskian-style interpretations for formulae in typed first-order logic, using the TPTP TF0 language. It further describes a technique and an implemented tool for verifying models using this representation, and a tool for visualizing interpretations. The research contributes to the advancement of automated reasoning technology for model finding, which has several applications, including verification.

1 Introduction

Historically, Automated Theorem Proving (ATP) has, as the name suggests, focused largely on the task of proving theorems from axioms – the derivation of conclusions that follow inevitably from known facts \cite{28}. The axioms and conjecture to be proved (and hence become a theorem) are written in an appropriately expressive logic, and the proofs are often similarly written in logic \cite{45}. In this work simply-typed first-order logic in the form of \cite{52, 30, 12}, whose expressive power is sufficient for a wide range of topics \cite{40}, is used (This work is also applicable to untyped first-order logic where terms have the type $\iota$ and formulae have the type $o$, and can also be generalized to higher-order logics.) In the last two decades the converse task of disproving conjectures, i.e., proving that a conjecture is not a theorem of the axioms, has become increasingly important. This process depends on finding an \textit{interpretation}, i.e., a structure that maps terms to domain elements and formulae to truth values. An interpretation that maps a formula to \textit{true} is a \textit{model} of the formula. A conjecture is disproved by finding an interpretation that is a model of the axioms, but is not a model of the conjecture, aka a \textit{countermodel} for the conjecture. A salient application area that harnesses this form of ATP is verification \cite{14}, where a countermodel is used to pinpoint the reason why a proof obligation fails, and correspondingly points to the location of the fault in the system being verified. Other applications of model finding include checking the consistency of an axiomatization \cite{32}, and finding a solution to a problem that is coded as a model finding problem \cite{53}. This work describes a (new) format for representing interpretations using a TPTP language - Sections 2 and 3.
In addition to ATP systems that produce interpretations (typically models), e.g., Paradox [11], Vampire [22], and Nitpick [9], there is a need for tools that support examination and use of interpretations. This paper considers the tasks of verifying models and visualizing interpretations, and describes new tools for these tasks - Sections 4 and 5.

Related Work: In [45] a TPTP format for interpretations with finite domains was defined, and it has been adopted by some ATP systems, e.g., Paradox and Vampire. The SMT-LIB standard [6] defines a format for model output, and commands to inspect models. SAT solvers generally output models as specified by the SAT competitions [20], in a simple format similar to the DIMACS input format [4]. Some individual model finding systems have defined their own formats for models, e.g., the output formats of Nitpick and Z3 [13].

Related work on model verification and interpretation visualization is sparse. In personal communications with members of the Vampire team it was revealed that Vampire can internally verify finite models in TPTP format by using the model formulae to evaluate the given formulae. This approach is limited to finite models. In personal communications with the developer of Paradox he explained his approach, which is to use a trusted model finder to show that the model formulae and the given formulae are together satisfiable. This shows that the model formulae are consistent with the given formulae, but does not verify the model – as the developer said, it is a “poor-man’s model verifier!”.

For interpretation visualization, the Mace4 model finder [25] outputs textual information about the models it finds, including the interpretation of constants as integers, and tables for the function and predicate symbols’ interpretations. The tables are naturally limited to symbols of arity up to two (which is just fine for algebras, where Mace4 is often applied). The only graphical visualization tool that has been found is described in [29], which provided (past tense – it is no longer available) a visualization of finite first-order interpretations as produced by Paradox. The visualization had some nice features, e.g., showing functions as constructor functions, and reducing the visual clutter when displaying relations with properties such as symmetry, transitivity, etc. In other ways that work was quite different from the visualization described in this work.

This paper is organized as follows: Section 2 introduces the TPTP World which provides the framework and languages used in this research. Section 3 discusses the nature of interpretations, and describes the new representation of interpretations using a TPTP language. Section 4 provides the theory for verifying models, and describes the implementation of that theory in a model verification tool. Section 5 introduces a novel way of visualizing interpretations, and proposes a tool for automating the visualization of interpretations written in the TPTP language. Section 6 concludes and discusses plans for future work.

2 The TPTP World and Languages

The TPTP World [40] is a well established infrastructure that supports research, development, and deployment of ATP systems. The TPTP World includes the TPTP problem library [37], the TSTP solution library [38], standards for writing ATP problems and reporting ATP solutions [45, 36], tools and services for processing ATP problems and solutions [38], and it supports the CADE ATP System Competition (CASC) [39]. Various parts of the TPTP World have been deployed in a range of applications, in both academia and industry. Since the first release of the TPTP problem library in 1993, many researchers have used the TPTP World as an
appropriate and convenient basis for ATP system research and development. Over the years the TPTP World has provided a platform upon which ATP users have presented their needs to ATP system developers, who have then adapted their ATP systems to the users’ needs. The web page https://www.tptp.org provides access to all components.

The TPTP language [41] is one of the keys to the success of the TPTP World. The language is used for writing both problems and solutions, which enables convenient communication between systems. Originally the TPTP World supported only first-order clause normal form (CNF) [46]. Over the years full first-order form (FOF) [37], typed first-order form (TFF) [44, 10], typed extended first-order form (TXF) [43], typed higher-order form (THF) [42, 21], and non-classical forms (NTF) \(^1\) [33] have been added. A general principle of the TPTP language is “we provide the syntax, you provide the semantics”. As such, there is no a priori commitment to any semantics for the languages, although in almost all cases the intended logic and semantics are well known. All the typed forms include constructs for arithmetic. TF0 [44], the monomorphic subform of TFF, is used in this work (see Section 2.1).

The top level building blocks of the TPTP language are annotated formulae. An annotated formula has the form:

```
language(name, role, formula, source, useful_info)
```

The languages supported are cnf (clause normal form), fof (first-order form), tff (typed first-order form), and thf (typed higher-order form). The role, e.g., axiom, lemma, conjecture, defines the use of the formula in an ATP system. In a formula, terms and atoms follow Prolog conventions – functions and predicates start with a lowercase letter or are ‘single quoted’, and variables start with an uppercase letter. The language also supports interpreted symbols, which either start with a $, e.g., the truth constants $true and $false, or are composed of non-alphabetic characters, e.g., integer/rational/real numbers such as 27, 43/92, -99.66. The logical connectives in the TPTP language are $!, $?, $\langle$, $\rangle$, $\&$, $\Rightarrow$, $\Leftarrow$, and $\Leftrightarrow$, for the mathematical connectives $\forall$, $\exists$, $\neg$, $\lor$, $\land$, $\Rightarrow$, $\Leftarrow$, and $\Leftrightarrow$ respectively. Equality and inequality are expressed as the infix operators $=$ and $\neq$. The source and useful_info are optional. Annotated formulae (using TF0) can be seen in Figures 1-5.

2.1 The TF0 Language

TF0 is a typed first-order language. The TF0 types are (i) the predefined types $i$ for individuals and $o$ for booleans; (ii) the predefined arithmetic types $int$, $rat$, and $real$; (iii) user-defined types declared to be of the kind $tType$. Every symbol is declared with a type signature: (i) individual types for variables; (ii) function types from non-boolean argument types to a non-boolean result type; (iii) predicate types from non-boolean argument types to a boolean result. The equality predicates $=$ and $\neq$ are ad hoc polymorphic over all types. Arithmetic predicates and functions are ad hoc polymorphic over the arithmetic types. Figures 1 and 2 are examples of problems in TF0. Their associated (counter)models are discussed in Section 3.

3 Interpretations

A Tarskian-style interpretation [47] of formulae in typed first-order logic consists of a non-empty domain of unequal elements for each type used in the formulae (just one domain for untyped logic), and interpretations of the function and predicate symbols with respect to the domains

\(^1\)There are many “non-classical logics”, including multi-valued logics [3], paraconsistent logics [27], relevance logics [2], etc. In this work we are interested in those that admit Kripke interpretation [23], e.g., modal logics [8].
% tff(human_type,type, human: $tType ).
tff(cat_type,type, cat: $tType ).
tff(jon_decl,type, jon: human ).
tff(garfield_decl,type, garfield: cat ).
tff(arlene_decl,type, arlene: cat ).
tff(normal_decl,type, normal: cat ).
tff(loves_decl,type, loves: cat > cat ).
tff(owns_decl,type, owns: ( human * cat ) > $o ).

tff(only_garfield_and_arlene_and_normal,axiom,
    ! [C: cat] : ( C = garfield | C = arlene | C = normal ) ).
tff(distinct_cats,axiom,
    ( garfield != arlene & arlene != normal & normal != garfield ) ).
tff(only_garfield_and_arlene_and_normal,axiom,
    ! [C: cat] : ( owns(jon,garfield) & ~ owns(jon,arlene) )).
tff(jon_owns_garfield_not_arlene,axiom,
    ! [C: cat] : is_descendant(A,D) => owns(jon,C) )

% Figure 1: A TF0 problem (with a finite countermodel)
https://raw.githubusercontent.com/GeoffsPapers/ModelVerificationLPAR/master/TFF_Finite.p

% tff(person_type,type, person: $tType ).
tff(bob_decl,type, bob: person ).
tff(child_of_decl,type, child_of: person > person ).
tff(descendant_transitive,axiom,
    ! [A: person,C: person,G: person] :
        ( is_descendant(A,C) & is_descendant(C,G) ) => is_descendant(A,G) ).
tff(child_is_descendant,axiom,
tff(all_have_child,axiom,

% Figure 2: A TF0 problem (with an infinite model)
https://raw.githubusercontent.com/GeoffsPapers/ModelVerificationLPAR/master/TFF_Infinite.p
$I \vdash \Phi$ means the interpretation $I$ is a model of the formula $\Phi$. An interpretation can normally interpret all expressions that can be written in the language of the formulae, but in some circumstances an interpretation can interpret only (at least) the given formulae; such an interpretation is a partial interpretation.

The domains of an interpretation may be finite or infinite. Interpretations with only finite domains are called finite interpretations, and interpretations with one or more infinite domains are called infinite interpretations. Finite domains are commonly explicitly enumerated, but can also take other forms, e.g., the finite Herbrand Universe of a Herbrand interpretation [17]. Infinite domains can take several forms, including being implicitly specified (e.g., some set of algebraic numbers, such as the integers), explicitly generated (e.g., terms representing Peano numbers), and the infinite Herbrand Universe of a Herbrand interpretation.

In addition to Tarskian-style interpretations that provide explicit symbol interpretation, a Herbrand interpretation can also be embodied in a saturation [5], i.e., a fixed point for a set of clauses at which further application of a complete inference system does not generate any new clauses. This approach is adopted in saturation-based ATP systems such as E [31], Prover9 [24], Vampire, and Zipperposition [51]. While the domain of a saturation is known to be the Herbrand Universe, there is no explicit symbol interpretation that can be used constructively by users. Saturations are thus a less useful form of interpretation. This work considers only Tarskian-style interpretations.

The notions of interpretations, models, partial interpretations, finite interpretations, Herbrand interpretations, etc., are captured in the SZS ontologies [36], as updated at https://www.tptp.org/cgi-bin/SeeTPTP?Category=Documents&File=SZSOntology

3.1 Representing Interpretations in TF0

As noted in Section 1, a TPTP format for interpretations with finite domains has previously been defined, and was been adopted by some ATP systems. Recently the need for a format for interpretations with infinite domains, and for a format for Kripke interpretations [23] of formulae written in the NTF language [33], led to the development of a new TPTP format for interpretations. The changes allow for multiple interpretations to be given in a single file, which, in the case of typed logics, share type declarations. The underlying principle is unchanged: interpretations are represented as formulae. This provides the basis for verification of models, as explained in Section 4.

The new format uses an interpretation formula. Examples of interpretation formulae can be seen in Figures 3 and 4, illustrating the components described next. An interpretation formula is a conjunction of three components:

- a conjunction of the domain specifications for the types in the given formulae: for each type a type-promotion function that converts domain elements to terms is used to keep the interpretation formula well-typed; each domain specification is a conjunction of:
  - the domain type, by a formula that makes the type-promotion function a surjection (unless it is unnecessary because the type is defined and is the same as the type in the given formulae, e.g., both are $\text{int}$);
  - the domain elements (unless implicit from their defined type): if the domain is finite this is a universally quantified disjunction of equalities whose right-hand sides are the terms; if the domain is infinite an existentially quantified formula that captures an infinite disjunction of equalities is used, e.g., for terms representing Peano numbers as the domain elements:
    \[ \forall I: \text{peano} \ (I = \text{zero}) \lor \exists P: \text{peano} \ (I = s(P)) \];
specification of the distinctness of the domain elements (unless implicit from their defined type);  
a formula making the type-promotion function an injection, which together with the surjectivity makes it a bijection.  
• interpretation of the function symbols, as equalities whose left-hand sides are formed from symbols applied to type-promoted domain elements, and whose right-hand sides are type-promoted domain elements;  
• interpretation of the predicate symbols, as literals formed from symbols applied to type-promoted domain elements; positive literals are true and negative literals are false.

The interpretation formula is preceded by the necessary type declarations:  
• the types in the given formulae (except defined types, e.g., $\text{int}$);  
• the types of the domains (except defined types);  
• the types of type-promotion functions;  
• the types of the domain elements.

This representation is also directly usable for untyped first-order logic, where all terms in the given formulae and the interpretation formula are of the same type – “individuals”. This obviates the need for type considerations, in particular type-promotion functions are not needed.

Figure 3 is a TF0 interpretation with finite domains – it is a countermodel for the problem in Figure 1. The comments show which parts of the formula specify what aspects of the interpretation. Figure 4 is a TF0 interpretation with an infinite domain – it is a model for the problem in Figure 2. Note that in Figure 4: the defined type $\text{int}$ is the domain type for the formula type person, so that there is no specification of the domain elements and their distinctness; universal quantification is used for the interpretation of function and predicate symbols for an infinite number of argument tuples; the interpretations of function and predicate symbols is not given for argument tuples with negative integers, i.e., this is an example of a partial interpretation.

4 Model Verification

ATP systems are complex pieces of software, implementing complex calculi, with the end goal being a sound implementation of a sound inference system whose output correctly corroborates the result obtained. The reality is that the complexity leads to incorrectness, and as such verification of ATP systems’ outputs is necessary. For theorem proving this means verifying the proof output [34], and for model finding this means verifying the model output. In the context of this work the model verification applies to the type declarations and the interpretation formula that represent the model found by the ATP system, and has (at least) the following aspects:

1. Are the type declarations and interpretation formula syntactically well-formed and semantically well-typed?
2. Is the interpretation formula satisfiable?
3. Does the interpretation formula correctly represent the interpretation found by the ATP system?
4. Is the interpretation represented by the interpretation formula a model for the given formulae?

These questions are answered as follows:
%----Types of the domains
%----The domain for human is d_human
%----The d_human elements are \{d_jon\}
& ! [DH: d_human] : ( DH = d_jon ) %----The type-promoter is a bijection
& ! [DH1: d_human,DH2: d_human] : ( d2human(DH1) = d2human(DH2) => DH1 = DH2 )
%----The domain for cat is d_cat
& ! [C: cat] : ? [DC: d_cat] : C = d2cat(DC) %----The d_cat elements are \{d_garfield,d_arlene,d_nermal\}
& ! [DC: d_cat] : ( DC = d_garfield | DC = d_arlene | DC = d_nermal ) & $\text{distinct}(d\_garfield,d\_arlene,d\_nermal)$ %----The type-promoter is a bijection
& ! [DC1: d_cat,DC2: d_cat] : ( d2cat(DC1) = d2cat(DC2) => DC1 = DC2 ) %----Interpret terms via the type-promoted domain
%----Interpret atoms as true or false
& (! owns(d2human(d_jon),d2cat(d_garfield)) & ! owns(d2human(d_jon),d2cat(d_arlene)) & ! owns(d2human(d_jon),d2cat(d_nermal)) )

Figure 3: A TF0 interpretation with a finite domain

https://raw.githubusercontent.com/GeoffsPapers/ModelVerificationLPAR/master/TFF_Finite.s
tff(person_type,type, person: $tType ).
tff(bob_decl,type, bob: person ).
tff(child_of_decl,type, child_of: person > person ).
tff(is_descendant_decl,type, is_descendant: ( person * person ) > $o ).
tff(int_to_person_decl,type, int_to_person: $int > person ).
tff(people,interpretation,
%----Domain for type person is the integers
%----The type promoter is a bijection (injective and surjective)
& ! [I1: $int,I2: $int] :
( int_to_person(I1) = int_to_person(I2) => I1 = I2 ) )
%----Mapping people to integers. Note that Bob’s ancestors will be interpreted
%----as negative integers.
& ( bob = int_to_person(0)
& ! [I: $int] : child_of(int_to_person(I)) = int_to_person($sum(I,1)) )
%----Interpretation of descendancy
& ! [A: $int,D: $int] :
( is_descendant(int_to_person(A),int_to_person(D)) <= $less(A,D) ) )).

Figure 4: A TF0 interpretation with an infinite domain
https://raw.githubusercontent.com/GeoffsPapers/ModelVerificationLPAR/master/TFF_Integer.s

1. This can be confirmed using standard parsing and type checking tools, e.g., [50, 18].

2. This can be empirically confirmed using a trusted model finder (in the same way the
GDV derivation verifier [34] uses the Otter system [26] as a trusted theorem prover).
Confirming that the interpretation formula is satisfiable is almost certainly much easier
than finding the model itself, so the system used to check the satisfiability can be weaker
and more trusted than the system that found the model.

3. This cannot be confirmed, as that representation is internal to the ATP system that found
the model.

4. In this work a “semantic” approach is taken, in which the given formulae Φ are proved
from the interpretation formula φ using a trusted theorem prover; φ is supplied as an
axiom, and Φ as the conjecture to be proved. This approach relies on the proof of
soundness below, which shows that if Φ can be proved from φ (written φ |= Φ), then the
interpretation I represented by φ is a model of Φ.

An implementation is available online as the AGMV tool in the SystemOnTSTP [35] web
interface https://www.tptp.org/cgi-bin/SystemOnTSTP. The tool input is the concatenation
of the problem and the interpretation. Figure 5 shows the verification problem for
the problem in Figure 2 and its model in Figure 4. The input to verify the finite counter-
model in Figure 3, for the problem in Figure 1, is https://raw.githubusercontent.com/
GeoffsPapers/ModelVerificationLPAR/master/TFF_Finite.sp.AGMV.p.

The proof of soundness is given here for a finite interpretation in untyped first-order logic,
where (as explained in Section 3.1) there is no need for type considerations. The proof for
typed first-order logic follows exactly the same pattern, but is technically complicated due to
the introduction of types and type promotion functions. The extension to infinite domains is quite simple after that, following Section 3.1.

**Proof**

Let $\Sigma$ be an untyped first-order language:

- $V_\Sigma$ - The variable symbols, starting in uppercase.
- $F_\Sigma$ - The function symbols with arity, in the form $f/n$.
- $P_\Sigma$ - The predicate symbols with arity, in the form $p/n$.

The formulae over $\Sigma$, $F(\Sigma)$, are defined as usual.

Let $I$ be an interpretation for $\Sigma$:

- $D_I$ - A finite set of unequal domain elements.
- $F_I$ - For each $f/n \in F_\Sigma$, a mapping $f_I : D^*_I \mapsto D_I$.
- $P_I$ - For each $p/n \in P_\Sigma$, a mapping $p_I : D^*_I \mapsto \{true, false\}$.
Recalling Section 3.1, an interpretation is represented by an *interpretation formula*, \( \varphi \). Let:

- \( D_\varphi \) be a set of fresh terms \( d_\varphi \), one for each \( d_I \in D_I \)
- \( D_{\varphi \rightarrow I} \) be the corresponding mapping from \( D_\varphi \) to \( D_I \)
- \( \Sigma_\varphi \) be the untyped first-order language:
  - \( \forall \in \Sigma_\varphi \)
  - \( \exists \in \Sigma_\varphi \)
  - \( \forall \in \Sigma_\varphi \)

\( \varphi \in F(\Sigma_\varphi) = D_\varphi \land D_\varphi \land F_\varphi \land P_\varphi ^\land \), where:

\[
\begin{align*}
D_\varphi ^\lor & = \forall X \bigvee_{d_\varphi \in D_\varphi } (X = d_\varphi ) \\
D_\varphi ^\land & = \bigwedge_{d_\varphi , e_\varphi \in D_\varphi , d_\varphi \neq e_\varphi } (d_\varphi \neq e_\varphi ) \\
F_\varphi ^\land & = \bigwedge_{f_\varphi , f_I , f_I \in F_I , f_\varphi = f_I (d_\varphi _I = d_\varphi \rightarrow d_I , i) = d_I , i , \neg (d_\varphi _I (d_\varphi _I = d_\varphi \rightarrow d_I , i) = d_I , i) \in f_I } \neg p(d_\varphi _I , i) \\
P_\varphi ^\land & = \bigwedge_{p_\varphi , p_I , p_I \in P_I , (d_\varphi _I (d_\varphi _I = d_\varphi \rightarrow d_I , i) = d_I , i) \in p_I } p(d_\varphi _I , i) \\
\end{align*}
\]

Let \( I_\varphi \) be an interpretation for \( \Sigma_\varphi \):

- \( D_{I_\varphi} = D_I \)
- \( F_{I_\varphi} = F_I \cup D_{\varphi \rightarrow I} \)
- \( P_{I_\varphi} = P_I \)

**Lemma.** \( I_\varphi \vdash \varphi \)

**Proof.** To prove \( I_\varphi \vdash \varphi \), prove \( I_\varphi \vdash D_\varphi ^\lor , I_\varphi \vdash D_\varphi ^\land , I_\varphi \vdash F_\varphi ^\land \) and \( I_\varphi \vdash P_\varphi ^\land \):

- For every \( d_{I_\varphi} \in D_{I_\varphi} \), or equivalently \( d_I \in D_I \):
  - There is a \( d_\varphi \in D_\varphi \) such that \( D_{\varphi \rightarrow I}(d_\varphi ) = d_I \)
  - \( (X = d_\varphi ) \in D_\varphi ^\lor \)
  - With \( X \) set to \( d_I \):
    \[ I_\varphi \vdash (d_I = d_\varphi ) \text{ if } d_I = D_{\varphi \rightarrow I}(d_\varphi ) \text{ if } d_I = D_{\varphi \rightarrow I}(d_\varphi ) \]
    which is true from the selection of \( d_\varphi \)

For every \( d_{I_\varphi} \in D_{I_\varphi} \), with \( X \) set to \( d_{I_\varphi} \), a disjunct in \( D_\varphi ^\lor \) is true, i.e., \( I_\varphi \vdash D_\varphi ^\lor \)
• For every \((d_\varphi \neq e_\varphi)\) in \(D_\varphi^\neq\):

\[
- I_\varphi \models (d_\varphi \neq e_\varphi) \text{ iff } F_{I_\varphi}(d_\varphi) \neq F_{I_\varphi}(e_\varphi) \text{ iff } D_{\varphi \rightarrow I_\varphi}(d_\varphi) \neq D_{\varphi \rightarrow I_\varphi}(e_\varphi) \text{ iff } d_I \neq e_I
\]

which is true from the definition of \(D_I\)

Thus every inequality in \(D_\varphi^\neq\) is true, therefore \(D_\varphi^\neq\) is true, i.e., \(I_\varphi \vdash D_\varphi^\neq\)

• For every \((f(d_{\varphi,i}) = d_\varphi)\) in \(F_\varphi^=\):

\[
- I_\varphi \models (f(d_{\varphi,i}) = d_\varphi) \text{ iff } F_{I_\varphi}(f(d_{\varphi,i})) = F_{I_\varphi}(d_\varphi) \text{ iff } f_I(D_{\varphi \rightarrow I_\varphi}(d_{\varphi,i})) = D_{\varphi \rightarrow I_\varphi}(d_\varphi) \text{ iff } f_I(d_{I,i}) = d_I
\]

which is true from the use of \(F_I\) in \(F_\varphi^=\)

Thus every equality in \(F_\varphi^=\) is true, therefore \(F_\varphi^=\) is true, i.e., \(I_\varphi \vdash F_\varphi^=\)

• For every (positive) \(p(d_{\varphi,i})\) in \(P_\varphi^p\):

\[
- I_\varphi \models p(d_{\varphi,i}) \text{ iff } P_{I_\varphi}(p(F_{I_\varphi}(d_{\varphi,i}))) \text{ iff } p_I(D_{\varphi \rightarrow I_\varphi}(d_{\varphi,i})) \text{ iff } p_I(d_{I,i})
\]

which is true from the use of \(P_I\) in \(P_\varphi^p\)

Thus every (positive) \(p(d_{\varphi,i})\) in \(P_\varphi^p\) is true. Analogously, every (negative) \(\neg p(d_{\varphi,i})\) in \(P_\varphi^p\) is false. Therefore \(P_\varphi^p\) is true, i.e., \(I_\varphi \vdash P_\varphi^p\)

\[\square\]

**Theorem.** Let \(\Phi \in F(\Sigma)\), \(I\) an interpretation for \(\Sigma\), and \(\varphi\) the interpretation formula for \(I\). If \(\varphi \models \Phi\) then \(I \models \Phi\).

**Proof.**

• If \(\varphi \models \Phi\) then \(I_\varphi \models \Phi\) because every model of \(\varphi\) is a model of \(\Phi\), and \(I_\varphi\) is a model of \(\varphi\) by the Lemma.

• \(I_\varphi \models \Phi\) iff \(I \models \Phi\) because \(\Phi\) contains no symbols from \(D_\varphi\), and \(I_\varphi\) is the same as \(I\) with respect to all other symbols.

• Thus if \(\varphi \models \Phi\) then \(I \models \Phi\).

\[\square\]
5 Interpretation Visualization

Proof visualization is well-established, with several tools available, e.g., Evonne [1] is an interactive proof visualization software for description logics; ProofTree [48] is a proof visualization tool focused on interactive theorem proving within Coq; Treehehe [7] was designed generically to visualize any proof tree but currently it supports only a handful of pre-existing proofs and does not allow users to visualize their own proofs; and the Interactive Derivation Viewer (IDV) [49] is a tool for visualization of TPTP format proofs. Interpretation visualization, however, has (to the knowledge of the authors) had minimal attention, as noted in Section 1. Visualization of interpretations is useful in areas such as teaching logic, debugging ATP systems, and understanding of a model.

A visualization for TF0 interpretations has been designed in this work, and an initial implementation is available as the IIV tool in SystemOnTSTP. IIV is built on top of IDV, and has benefited from the mature state of IDV. IDV was originally a Java applet, but has since been ported to HTML/JavaScript using GraphViz [15] for the layout and rendering. IIV has benefited from the mature state of IDV. The implementation is “initial” because it is fully automated for only finite TF0 and FOF interpretations; for infinite interpretations different components of the interpretation formula have to be manually extracted into separate annotated formulae, to mimic a derivation that IDV can render.

Figure 6 is the visualization of the finite countermodel in Figure 3. The top row of inverted triangles are the types in the given formulae, while the bottom row of inverted triangles are the types of the domains. The inverted houses are the function and predicate symbols, and the successive rows of ovals are the successive domain element arguments used to specify the symbols’ interpretations. Finally, the row of houses and boxes are the interpretations of the symbols applied to those arguments; houses for domain elements and boxes for truth values. Paths from leaf type nodes to root type nodes show the interpretation of symbols and the domain elements. For example, in Figure 6 the result type of loves is cat, and loves(darlene) is interpreted as dgarfield, which is of type dcat in the interpretation formula.

IIV has interactive features: In Figure 6 the cursor is hovering over the dnormal node on the path from owns to $false, showing that owns(djon,dnormal) is interpreted as $false. The nodes above are increasingly darker red (grey if printed) up to the $o node that is the result type of owns, and increasingly darker blue down to the $o node that is the type of $true. This highlighting provides easy focus on the interpretation of chosen symbols, e.g., hovering over inverted house nodes shows what symbols applied to what domain elements are interpreted as which domain elements and boolean values, and hovering over oval nodes shows how different domain elements affect the interpretation of symbols. This visualization is available in IIV using https://raw.githubusercontent.com/GeoffsPapers/ModelVerificationLPAR/master/TFF_Finite.s as the “URL to fetch from”, selecting IIV 0.0 as the “System”, and clicking the “Process Solution” button.

Figure 7 is the visualization of the infinite model in Figure 4. Here (universally quantified) variables are used to represent an infinite number of domain elements, and built-in arithmetic predicates are used to compute symbols’ mappings. The cursor is hovering over the X:$int node, showing how child_of(X) is interpreted as $sum(X,1).

6 Conclusion

This paper describes the new TPTP format for representing Tarskian-style interpretations for formulae in typed first-order logic, using the TPTP TF0 language. It further describes a
technique and an implemented tool for verifying models using this representation, and a tool for visualizing interpretations. The research contributes to the advancement of automated reasoning technology for model finding, which has several applications, including verification.

Currently this work is being extended to Kripke interpretations for formulae in non-classical typed first-order logic [33], using the TPTP NX0 language [41]. The tool to translate interpretation formulae to the format required for input to the IIV tool is being extended to infinite interpretations. Further inspiration might also lead to improvements to IIV’s visualizations, especially for more complex infinite interpretations.

References


Figure 7: Visualization of the interpretation in Figure 4


