An Abstraction-Refinement Framework for Reasoning with Large Theories

Julio Cesar Lopez Hernandez and Konstantin Korovin

The University of Manchester, School of Computer Science
{lopezhej,korovin}@cs.man.ac.uk

Abstract. In this paper we present an approach to reasoning with large theories which is based on the abstraction-refinement framework. The proposed approach consists of the following approximations: the over-approximation, the under-approximation and their combination. We present several concrete abstractions based on subsumption, signature grouping and argument filtering. We implemented our approach in a theorem prover for first-order logic iProver and evaluated over the TPTP library.

Keywords: automated reasoning, large theories, abstraction-refinement

1 Introduction

Efficient reasoning with large theories is one of the main challenges in automated theorem proving arising in many applications ranging from reasoning with ontologies to proof assistants for mathematics. Current methods for reasoning with large theories are based on different axiom selection methods. Some of them are based on the syntactic or semantic structure of the axioms and conjecture formulas [15, 30]. These methods select relevant axioms based on syntactic or semantic relationship between axioms and conjectures. Other methods for axiom selection use machine learning to take advantage of previous knowledge about proved conjectures [16, 32, 33]. What those methods have in common are two phases of the whole process for proving a conjecture: one is the axiom selection phase, and the other one is the reasoning phase. Those phases are performed in a sequential way. First, the axiom selection takes place, then using the selected axioms the reasoning process starts.

Our proposed approach based on abstraction-refinement framework [8] has the purpose of interleaving the axioms selection and reasoning phases, having a more dynamic interaction between them. This proposed approach encompasses two ways for approximating axioms: one is called over-approximation and the other one under-approximation. Those approximations are combined to converge more rapidly to a proof if it exists or to a model otherwise. There are a number of related works which consider different specific types of under and/or over approximations in different contexts [1, 5, 7, 9, 12, 19, 22–24, 31]. Nevertheless, abstraction-refinement is largely overlooked in state-of-the-art automated theorem provers, with an exception of SPASS which was extended with

abstraction-refinement into a very specialised decidable fragment to approximate general first-order reasoning [31]. Another relevant example is the Inst-Gen calculus [19] which under-approximates first-order formulas by propositional/ground abstractions and refines these approximations by model-guided instantiations. In the SMT setting, ground approximations are used in conflict and model-based instantiation methods [11, 25]. In higher-order logic, over-approximations are used for efficient encodings into first-order logic [2–4], propositional logic [6] and also in higher-order patterns [10].

In this paper we take a pragmatic approach. Instead of targeting a specific decidable fragment as an abstract domain we use abstraction-refinement to simplify problems by different over and under approximations and their combinations. We present a general abstraction-refinement framework for refutation theorem proving which allows one to compare and combine different abstractions. Our framework is general enough to represent abstractions not only within the same language but also abstractions that extend or modify the language, in particular abstractions based on signature transformations. We present a number of concrete abstractions based on subsumption, signature grouping and argument filtering and discuss their combinations. In this paper we consider many-sorted first-order logic in the context of first-order theorem proving but the approach is applicable to SMT as well. [1]

2 Abstraction Functions and Refinements

Let us consider a set of formulas $\mathcal{F}$ which we call a *concrete domain* and a set of formulas $\hat{\mathcal{F}}$ which we will call an *abstract domain*. For example $\mathcal{F}$ can be the set of all first-order formulas and $\hat{\mathcal{F}}$ can be a fragment of first-order logic. Concrete and abstract domains can coincide.

An *abstraction function* is a mapping $\alpha : \mathcal{F} \mapsto \hat{\mathcal{F}}$. When there is no ambiguity we will call an abstraction function just an abstraction of $\mathcal{F}$. The identity function is an abstraction which will be called the *identity abstraction* α_{id}.

A *concretisation function* for α is the inverse mapping $\gamma : \hat{\mathcal{F}} \mapsto 2^{\mathcal{F}}$, i.e., $\gamma(\hat{F}) = \{F \mid \alpha(F) = \hat{F}\}$ for $\hat{F} \in \hat{\mathcal{F}}$.

An abstraction α is called *over-approximating abstraction* (wrt. refutation) if for every $F \in \mathcal{F}$, $F \models \bot$ implies $\alpha(F) \models \bot$. An abstraction α is called *under-approximating abstraction* (wrt. refutation) if for every $F \in \mathcal{F}$, $\alpha(F) \models \bot$ implies $F \models \bot$.

We can compose abstractions as mappings. In particular, if $\alpha_1 : \mathcal{F} \mapsto \mathcal{F}_1$ and $\alpha_2 : \mathcal{F}_1 \mapsto \mathcal{F}_2$ then $\alpha_1\alpha_2$ is an abstraction of $\mathcal{F}$.

Proposition 1. *Composition of over-approximating abstractions is an over-approximating abstraction. Likewise, composition of under-approximating abstractions is an under-approximating abstraction.*

In this paper we will define several atomic abstractions and we use this proposition to compose them to obtain a large range of combined abstractions.

[1] Preliminary version of this work was presented at the IWIL workshop [13].

We define an ordering on abstractions $\sqsubseteq$ called *abstraction refinement ordering* as follows: $\alpha \sqsubseteq \alpha'$ if for all $F \in \mathcal{F}$, $\alpha(F) \models \bot$ implies $\alpha'(F) \models \bot$. Two abstractions are *equivalent*, denoted by $\alpha \equiv \alpha'$ if $\alpha \sqsubseteq \alpha'$ and $\alpha' \sqsubseteq \alpha$. The strict part $\sqsubset$ of $\sqsubseteq$ is defined as $\alpha \sqsubset \alpha'$ if $\alpha \sqsubseteq \alpha'$ and $\alpha \not\equiv \alpha'$. An abstraction is *precise* if it is equivalent to the identity abstraction. An example of a non-trivial precise abstraction can be obtained by renaming function and predicate symbols. We have that every over-approximating abstraction α_s is above and every under-approximation abstraction α_w is below the identity abstraction wrt. the abstraction refinement ordering, i.e., $\alpha_w \sqsubseteq \alpha_{id} \sqsubseteq \alpha_s$.

Weakening abstraction refinement of an over-approximating abstraction α is an abstraction α' which is below α and above the identity abstraction in the abstraction refinement ordering, i.e., $\alpha_{id} \sqsubseteq \alpha' \sqsubseteq \alpha$. *Strengthening abstraction refinement* of an under-approximating abstraction α is an abstraction α' which is above α and below the identity abstraction in the abstraction refinement ordering, i.e., $\alpha \sqsubseteq \alpha' \sqsubseteq \alpha_{id}$.

An *over-approximation abstraction-refinement process* is a possibly infinite sequence of weakening abstraction refinements $\alpha_0, \ldots, \alpha_n, \ldots$ such that $\alpha_{id} \sqsubseteq \ldots \sqsubseteq \alpha_n \sqsubseteq \ldots \sqsubseteq \alpha_0$. Similar, an *under-approximation abstraction-refinement process* is a possibly infinite sequence of strengthening abstraction refinements $\alpha_0, \ldots, \alpha_n, \ldots$ such that $\alpha_0 \sqsubseteq \ldots \sqsubseteq \alpha_n \sqsubseteq \ldots \sqsubseteq \alpha_{id}$.

3 Over-Approximation Procedure

We use ATP_S to denote an automated theorem prover which is sound but possibly incomplete (wrt. refutation) [14]. On the other hand, we use ATP_C to make a reference to an automated theorem prover which is complete but not necessary sound [5, 22]. Hence, if ATP_S returns UNSAT then the conjecture is proved and if ATP_C returns SAT then the conjecture is disproved. The purpose of these ATPs is to prove or disprove conjectures more efficiently than a sound and complete ATP but with a possible loss of precision.

We consider a theory A which is a collection of axioms which we call *concrete axioms* and a set of formulas $\hat{A}^s$ called *abstract axioms*. We will assume that the negation of the conjecture is included in A, so proving the conjecture corresponds to proving unsatisfiability of A.

The *over-approximating procedure* starts by applying an over-approximating abstraction function α_s to A, to obtain an abstract representation of axioms $\hat{A}^s$, $\hat{A}^s = \alpha_s(A)$. First, the procedure tries to prove unsatisfiability of the abstract axioms $\hat{A}^s$ using an ATP_C. If ATP_C proves unsatisfiability of $\hat{A}^s$, the procedure extracts an abstract unsat core $\hat{A}^s_{uc}$ from $\hat{A}^s$, which can be obtained by, e.g., collecting all axioms involved in the abstract proof. Next, the procedure tries to prove unsatisfiability of the concretisation of the abstract unsat core $A_{uc} = \gamma_s(\hat{A}^s_{uc})$ using ATP_S. If the ATP_S proves unsatisfiability of A_{uc}, the process stops as this proves unsatisfiability of A. Otherwise, if A_{uc} is shown to be satisfiable, the set of axioms A is abstracted using a new abstraction α'_s obtained by weakening abstraction refinement of α_s. In practice, the refinement procedure

refines α_s until $\alpha'_s(A_{uc})$ becomes satisfiable, which is always possible as at this point we assume A_{uc} is satisfiable. The procedure is repeated utilising the refined set of abstract axioms. This loop finishes when the conjecture is proved or disproved or the time limit of the whole procedure is reached. The diagram of the over-approximating procedure is shown in Figure 1.

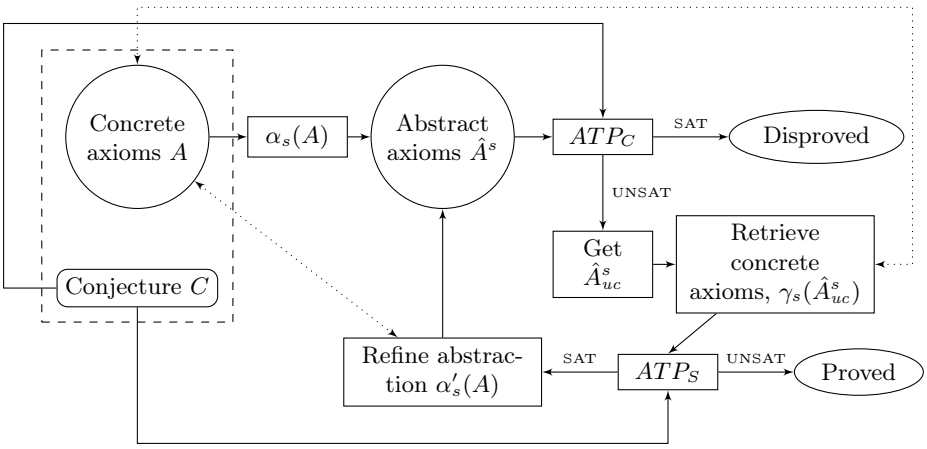

Fig. 1. The over-approximation procedure

The main parameters of this procedure are an over-approximating abstraction function and weakening abstraction refinement.

Next we define several concrete over-approximating abstractions and discuss abstraction refinement for these abstractions.

3.1 Subsumption-based Abstraction

In this section we present abstraction-refinement based on subsumption. Informally, we partition concrete axioms based on joint literal occurrences and for each partition we define an abstract clause which subsumes all clauses in the partition.

We define the initial abstraction of A as follows. With each set of clauses A', we associate a literal ℓ_k in A' which we call a partition literal for A'. An initial partition of A is defined as $A = A^{\ell_1^+} \cup A^{\ell_1^-}$ where ℓ_1 is a partition literal for A, members of $A^{\ell_1^+}$ are all clauses containing ℓ_1 and $A^{\ell_1^-} = A \setminus A^{\ell_1^+}$. We recursively continue partitioning $A^{\ell_1^-}$ in the same way until we obtain the empty set. The result of this process is the following partition of A:

$$A = \bigcup_{k=1}^{n} A^{\ell_1^- \ldots \ell_{k-1}^- \ell_k^+},$$

where ℓ_k is the partition literal for $A^{\ell_1^- \ldots \ell_{k-1}^-}$, we assume $A^{\ell_1^- \ldots \ell_n^-}$ is empty and $A^{\ell_1^- \ldots \ell_{k-1}^- \ell_k^+} = A^{\ell_1^+}$ for $k = 1$.

For each partition $A^{\ell_1^- \ldots \ell_{k-1}^- \ell_k^+}$ literals $\ell_1, \ldots, \ell_{k-1}$ do not occur in any clause in the collection and ℓ_k occurs in all clauses in $A^{\ell_1^- \ldots \ell_{k-1}^- \ell_k^+}$. Figure 2 shows an example of such partition.

We say that ℓ_k is a *leading literal* in $A^{\ell_1^- \ldots \ell_{k-1}^- \ell_k^+}$ and each leading literal is the abstraction of their corresponding set. These abstractions form the set of abstract axioms $\hat{A}^s$. In practice, we can select the leading literal based on a heuristic criteria, e.g., the number of occurrences of a literal in the clause set.

Example 1. Consider the following set of concrete clauses A and its partition consisting of $A^{\ell_1^+}$, $A^{\ell_1^- \ell_2^+}$ and $A^{\ell_1^- \ell_2^- \ell_3^+}$. Where the leading literals are ℓ_1, ℓ_2, ℓ_3 and they form the abstract set of clauses $\hat{A}^s$, $\hat{A}^s = \{\ell_1, \ell_2, \ell_3\}$.

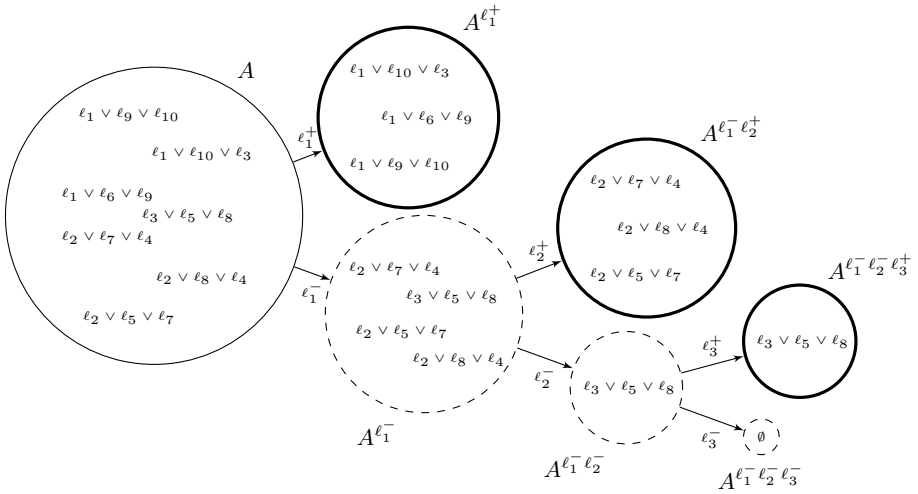

Fig. 2. Partitions of A are in bold

The mapping from sets of the form $A^{\ell_1^- \ldots \ell_{k-1}^- \ell_k^+}$ to the leading literals gives us the abstraction function α_s, which is defined as

$$\alpha_s(D) = \ell_k \text{ for } D \in A^{\ell_1^- \ldots \ell_{k-1}^- \ell_k^+}.$$

Consequently, the concretisation function γ_s is defined as

$$\gamma_s(\ell_k) = A^{\ell_1^- \ldots \ell_{k-1}^- \ell_k^+}.$$

We use the set of abstract axioms to try to prove a conjecture. If the conjecture is proved, we consider the abstract axioms from the unsat core. Those

abstract axioms are refined and then replaced by their refined versions. Then, the proving process is repeated using the refined set of abstract axioms. This process continues until we get a concrete proof of the conjecture.

The refinement of abstract axioms will be defined further in this section, but first consider the following definitions. During the refinement process we will partition A into sets of the form A^σ where σ is a sequence of signed literals $\sigma = \ell_1^{s_1} \dots \ell_n^{s_n}$, where s_j is either $+$ or $-$ for $1 \leq j \leq n$. A literal $\ell_j^{s_j}$ occurs in all clauses of A^σ if $s_j = +$ and does not occur in any of the clauses in the set if $s_j = -$.

In figure 2, the leaves different to the empty set are the partition of A and they have the form A^σ. The set of literals in σ with positive signs is defined as $\sigma^+ = \{\ell : \ell^s \in \sigma \text{ and } s \text{ is } +\}$. The set A^σ is abstracted with the clause $C^{\sigma^+} = \bigvee_{\ell \in \sigma^+} \ell$. Therefore, the abstraction function α_s is defined as

$$\alpha_s(D) = C^{\sigma^+},$$

where $D \in A^\sigma$. Then, concretisation function is $\gamma_s(C^{\sigma^+}) = A^\sigma$. The set A^σ is fully concretised if $A^\sigma = \{C^{\sigma^+}\}$.

For a set of clauses A', let $\mathcal{L}(A')$ denote the set of all literals occurring in clauses in A'. The refinement process is applied to an unsat core $\hat{A}^s_{uc}$ consisting of abstract clauses. The refinement process subpartitions one of $A^\sigma = \gamma_s(C^{\sigma^+})$, where $C^{\sigma^+} \in \hat{A}^s_{uc}$, such that A^σ is not fully concretised. Let $\sigma = \ell_1^{s_1} \dots \ell_k^{s_k}$. This process starts by selecting a new partition literal ℓ_k such that

$$\ell_{k+1} \in \mathcal{L}(A^\sigma) \setminus \sigma^+.$$

Note, that since A^σ is not fully concretised, $\mathcal{L}(A^\sigma) \setminus \sigma^+$ is not empty. Using the literal ℓ_{k+1}, we obtain the starting partition $A^\sigma = A^{\sigma\ell^+_{k+1}} \cup A^{\sigma\ell^-_{k+1}}$. Then, we continue recursively partitioning $A^{\sigma\ell^-_{k+1}}$ as before until we obtain the empty set. The result of this recursive process is the partition of A^σ defined as follows:

$$A^\sigma = \bigcup_{j=1}^{m} A^{\sigma\ell^-_{k+1}\dots\ell^-_{k+j-1}\ell^+_{k+j}},$$

where ℓ_{k+j} is the partition literal for $A^{\sigma\ell^-_{k+1}\dots\ell^-_{k+j-1}\ell^+_{k+j}}$ and $A^{\sigma\ell^-_{k+1}\dots\ell^-_{k+m-1}\ell^-_{k+m}}$ is the empty set. Denote $\sigma_i = \sigma\ell^-_{k+1} \dots \ell^-_{k+j-1}\ell^+_{k+j}$ for $1 \leq i \leq m$. Then refined abstraction α'_s is defined as:

$$\alpha'_s(D) = \begin{cases} C^{\sigma_i^+} & \text{if } D \in A^{\sigma_i} \text{ for some } 1 \leq i \leq m, \\ \alpha_s(D) & \text{if } D \notin A^\sigma. \end{cases}$$

An example of this refinement is shown in Figure 3, where the refined abstraction of A consists of $\{\ell_1, \ell_3, \ell_2 \vee \ell_4, \ell_2 \vee \ell_5\}$.

Let us note that the subsumption abstraction is an over-approximation abstraction and subsumption abstraction refinement is a weakening abstraction refinement, in particular, $\alpha_{id} \sqsubseteq \alpha'_s \sqsubseteq \alpha_s$.

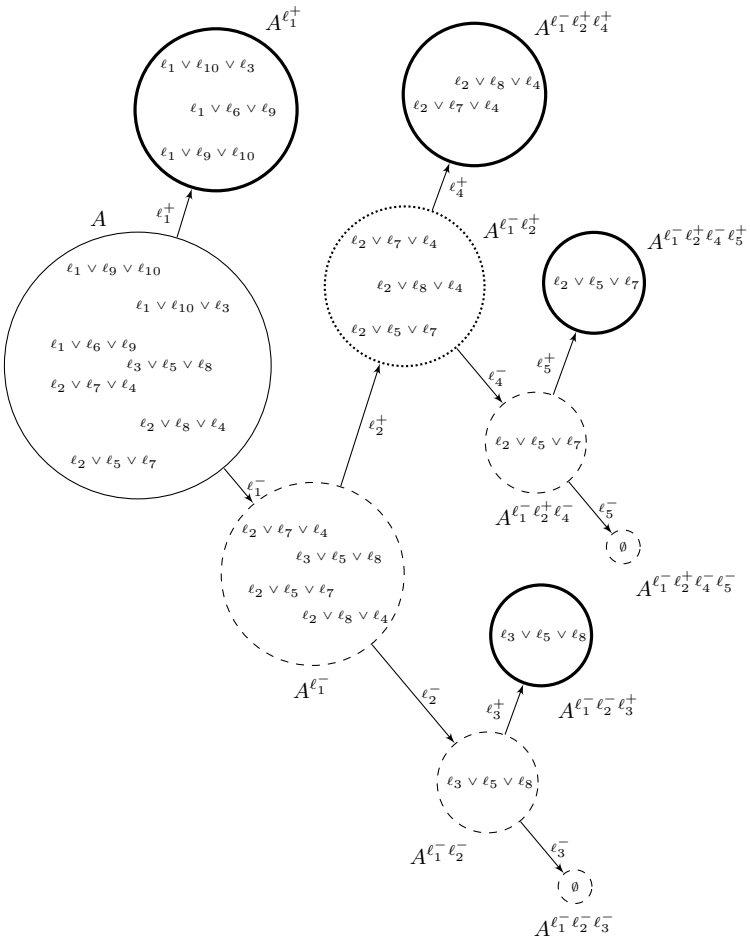

Fig. 3. Refinement of $A^{\ell_1^- \ell_2^+}$ (dotted circle); partitions of A are in bold

3.2 Generalisation Abstraction

In the generalisation abstraction we abstract clauses with their generalisations. A clause D is a generalisation of a clause C if $C = D\sigma$ for a substitution σ. Generalisation ordering on clauses can be defined as $C \sqsubseteq_g D$ if $C = D\sigma$. A generalisation abstraction α_g is a function that maps clauses to their generalisations, so we have $C \sqsubseteq_g \alpha_g(C)$. One example of the generalisation abstraction would be replacing certain non-variable terms by variables. For example, using a generalisation abstraction one can abstract the set of clauses into the Effectively PRopositional (EPR) fragment. Another abstraction strategy can be based on targeting inference positions eligible for superposition.

Example 2. Consider the following set of clauses:

$$S = \{p(g(x), g(x)) \vee q(f(g(x))); g(f(f(x))) \simeq g(f(x))\}.$$

A possible generalisation abstraction of S can be:

$$\alpha_g(S) = \{p(x, x) \vee q(f(x)); g(f(x)) \simeq g(x)\}.$$

Let us note that, e.g., superposition inference is applicable from the second into the first clause in S under any simplification ordering, which can easily lead to non-termination. On the other hand, there is no eligible superposition inferences between two abstracted clauses due to abstraction of terms headed with g in the first clause and the fact that superposition is not applied into the variable positions.

The generalisation abstraction refinement α' of α can be based on restoring abstracted terms in abstract clauses from the unsat core, i.e., $C \sqsubseteq_g \alpha'_g(C) \sqsubseteq_g \alpha_g(C)$ for $C \in \hat{A}^s_{uc}$ and $\alpha'_g(C) = \alpha_g(C)$ for $C \notin \hat{A}^s_{uc}$. We note that the generalisation abstraction is an over-approximation abstraction and generalisation abstraction refinement is a weakening abstraction refinement, in particular, $\alpha_{id} \sqsubseteq \alpha'_g \sqsubseteq \alpha_g$.

In practice, the generalisation abstraction can be naturally combined with the subsumption abstraction, by first generalising and then applying the subsumption abstraction.

3.3 Argument Filtering Abstraction

In this section we present the argument filtering abstraction. Informally, argument filtering abstraction is based on removing certain arguments in signature symbols.

Consider a signature Σ consisting of predicate and function symbols. We will represent argument selection using bit-vectors. Consider a bit-vector bv. We denote the length of bv by $|bv|$, the number of 1s in bv by $|bv|_1$ and 0s by $|bv|_0$. Let $\bar{\mathbf{1}}_n$, $\bar{\mathbf{0}}_n$ denote bit-vectors of length n, consisting of 1s and 0s, respectively. Let $\mathcal{B}_n$ denote the set of all bit-vectors of length n. We will omit index n when the bit-vector length is clear from the context or irrelevant.

With each signature (i.e., predicate or function) symbol f of arity n and a bit-vector bv of length n we associate an *abstract symbol* f^{bv} with the arity $|bv|_1$. An abstract domain for a signature symbol f, denoted $f^{\mathcal{B}}$ is the set of all abstract symbols f^{bv}, where $|bv| = \mathit{arity}(f)$. An abstract signature is defined as $\Sigma^{\mathcal{B}} = \cup_{f \in \Sigma} f^{\mathcal{B}}$. *A signature abstraction* is a function: $\alpha_f : \Sigma \mapsto \Sigma^{\mathcal{B}}$ such that $\alpha_f(f) \in f^{\mathcal{B}}$.

A signature abstraction can be extended to terms and atoms recursively:

$$\alpha_f(t) = \begin{cases} x & \text{if } t = x, \\ f^{bv}(\alpha_f(t_{i_1}), \ldots, \alpha_f(t_{i_k})) & \text{if } t = f(t_1, \ldots, t_n), \alpha_f(f) = f^{bv}, \text{ and} \\ & bv(i) = 1 \text{ iff } i \in \{i_1, \ldots, i_k\}. \end{cases}$$

In turn, α_f is extended to clauses and sets of clauses in an obvious way by applying α_f to atoms.

If we abstract every signature symbol f to $f^{\bar{\mathbf{1}}}$ then we obtain a precise abstraction, i.e., equivalent to the identity abstraction. Therefore, w.l.o.g., we will identify every signature symbol f with its $f^{\bar{\mathbf{1}}}$ abstraction.

Let us consider some special cases. If we abstract every predicate symbol p to $p^{\bar{\mathbf{0}}}$ then we obtain a *pure propositional abstraction*, which we denote α_f^{prop}. If we abstract every function symbol f to $f^{\bar{\mathbf{0}}}$ and every predicate symbol p to $p^{\bar{\mathbf{1}}}$ then we obtain an *EPR abstraction*, which we denote α_f^{EPR}. If we abstract every signature symbol f to $f^{\bar{\mathbf{1}}}$ then we obtain a precise abstraction, i.e., equivalent to the identity abstraction.

Example 3. Let us consider the following set of clauses

$$S = \{p(x, f(x, g(y))) \vee \neg p(c, x); \neg p(g(f(x, y)), g(y)); p(c, x)\}.$$

Then pure propositional abstraction will result in the following set of clauses:

$$\alpha_f^{prop}(S) = \{p^{\bar{\mathbf{0}}} \vee \neg p^{\bar{\mathbf{0}}}; \neg p^{\bar{\mathbf{0}}}; p^{\bar{\mathbf{0}}}\},$$

which is unsatisfiable. One the other hand the EPR abstraction is:

$$\alpha_f^{EPR}(S) = \{p(x, f^{\bar{\mathbf{0}}}) \vee \neg p(c, x); \neg p(g^{\bar{\mathbf{0}}}, g^{\bar{\mathbf{0}}}); p(c, x)\}.$$

It is easy to see that the EPR abstraction is satisfiable and therefore the original set of clauses is also satisfiable.

In order to define abstraction-refinement we introduce a partial ordering on abstract symbols: $f^{bv_0} \sqsubseteq_{af} f^{bv_1}$ iff $bv_1(i) \leq bv_0(i)$, for all $0 \leq i < arity(f)$. Then we extend this ordering on abstractions by defining $\alpha_f^0 \sqsubseteq_{af} \alpha_f^1$ iff $\alpha_f^0(f) \sqsubseteq_{af} \alpha_f^1(f)$ for all $f \in \Sigma$. We call $\sqsubseteq_{af}$ *argument filtering ordering.* The top element in this ordering is the pure propositional abstraction.

The following proposition implies that argument filtering abstraction is an over-approximation abstraction and abstraction refinement based on the argument filtering ordering is a weakening abstraction refinement.

Proposition 2. *The argument filtering ordering is compatible with the abstraction refinement ordering, i.e., if* $\alpha_f^0 \sqsubseteq_{af} \alpha_f^1$ *then* $\alpha_f^0 \sqsubseteq \alpha_f^1$. *Moreover, every argument filtering abstraction is above the identity abstraction, i.e.,* $\alpha_{id} \sqsubseteq \alpha_f$.

In the example above, the EPR abstraction is a refinement of the propositional abstraction. In practice, one can start with a propositional or EPR abstraction and define the weakening refinement process by restoring arguments of abstract symbols occurring in the unsat core, as described in the Section 3.

Abstracting variable dependencies. Let us observe how argument filtering can be used to abstract variable dependencies. As an example we consider clause splitting without backtracking [26], which can be defined as follows. Given a clause $C(\bar{x}, \bar{y}) \vee D(\bar{x}, \bar{z})$ one can split this clause into two clauses by introducing a fresh splitting predicate over joint variables $sp(\bar{x})$ and replacing this clause with two clauses $C(\bar{x}, \bar{y}) \vee sp(\bar{x})$ and $\neg sp(\bar{x}) \vee D(\bar{x}, \bar{z})$. In this way the splitting predicate represents variable dependencies between different subclauses. We can abstract such variable dependencies by restricting argument filtering abstraction-refinement to the splitting predicates. In the same way we can target formula definitions introduced during clausification and Skolem functions which encode existential variable dependencies.

3.4 Signature Grouping Abstraction

Consider a finite signature Σ and let $\mathcal{T}$ be the set of all types of symbols in Σ. In many-sorted first-order logic, a type of a symbol can be represented as a sequence of sorts in a standard way. We partition Σ into groups $\Sigma = \bigcup_{\tau \in \mathcal{T}} \Sigma_\tau$, such that symbols in Σ_τ are all symbols in Σ of type τ. With each non-empty subset of $\sigma_\tau \subseteq \Sigma_\tau$ we associate an abstract symbol f^{σ_τ} of type τ. The abstract signature Σ^S is defined as the union of all abstract symbols.

Consider partitioning Σ into groups $\Sigma = \cup_{i=1}^{n} \sigma_i$, such that all symbols in σ_i have the same type. We define a *signature grouping abstraction* α_{sig} as a function: $\alpha_{sig} : \Sigma \mapsto \Sigma^S$ such that $\alpha_{sig}(f) = f^{\sigma_i}$ if $f \in \sigma_i$ for some $1 \leq i \leq n$. In a similar way to Section 3.3, we extend α_{sig} to an abstraction over terms, atoms and clauses. We can also define an ordering on abstract symbols: $f^{\sigma_0} \sqsubseteq_{sig} f^{\sigma_1}$ iff $\sigma_0 \subseteq \sigma_1$ and extend this ordering to abstractions: $\alpha^0_{sig} \sqsubseteq_{sig} \alpha^1_{sig}$ iff $\alpha^0_{sig}(f) \sqsubseteq_{sig} \alpha^1_{sig}(f)$ for all $f \in \Sigma$. We call $\sqsubseteq_{sig}$ the *signature grouping ordering.* Let us note that the top element in this ordering is the abstraction corresponding to the maximal partitioning $\Sigma = \bigcup_{\tau \in \mathcal{T}} \Sigma_\tau$ and the bottom element is a precise abstraction corresponding to the partitioning into singleton sets.

Example 4. Consider the following set of clauses over a signature consisting of a single non-Boolean sort:

$$\{q(f(c)) \vee p(f(c)); \neg p(f(x)) \vee s(g(z), f(a)); \neg p(g(x)) \vee r(f(z), g(a)); \neg r(x, y)\},$$

we can group symbols of the same type such as q and p which are replaced by q'. Predicates s and r are replaced by s'; functions symbols f and g are replaced by f'. The resulting abstract set is:

$$\{q'(f'(c)); \neg q'(f'(x)) \vee s'(f'(z), f'(a)); \neg s'(x, y)\}.$$

This abstraction is unsatisfiable and we can refine it by concretising certain abstract symbols occurring in the unsat core, e.g.,

$$\{q(f'(c)) \vee p(f'(c)); \neg p(f'(x)) \vee s'(f'(z), f'(a)); \neg s'(x, y)\},$$

where q' is concretised.

Proposition 3. *The signature grouping ordering is compatible with the abstraction refinement ordering, i.e., if* $\alpha^0_{sig} \sqsubseteq_{sig} \alpha^1_{sig}$ *then* $\alpha^0_{sig} \sqsubseteq_{sig} \alpha^1_{sig}$*. Moreover, every signature grouping abstraction is above the identity abstraction, i.e.,* $\alpha_{id} \sqsubseteq \alpha_{sig}$*.*

Let us note that signature grouping can be naturally combined with the argument filtering abstraction. In particular, argument filtering can reduce symbol types which in turn can be used to produce larger groups of abstract symbols.

4 Abstraction by Under-Approximation

The process starts by applying the weakening abstraction function to the set of concrete axioms A, $\hat{A}^w = \alpha_w(A)$. This set $\hat{A}^w$ of weaker axioms is used to prove the conjecture, using an ATP_S. If the conjecture is proved the procedure stops and provides the proof. Otherwise, a model I of $\hat{A}^w$ and the negated conjecture is obtained. This model is used to refine the set of weaker axioms $\hat{A}^w$. During this refinement (strengthening abstraction refinement), the procedure tries to find a set of axioms $\breve{A}$ that turns the model into a countermodel but are still implied by A, i.e., $I \not\models \breve{A}$ and $A \models \breve{A}$. If the set of axioms $\breve{A}$ is empty, $\breve{A} = \emptyset$, the procedure stops and disproves the conjecture. Otherwise, the obtained set of axioms is added to the set of weaker axioms, $\hat{A}^w := \hat{A}^w \cup \breve{A}$. Using this new set of abstract axioms $\hat{A}^w$, another round for proving the conjecture starts. The process finishes when the conjecture is proved or disproved or the time limit for the quest of a proof is reached. The diagram of this procedure is shown in Figure 4.

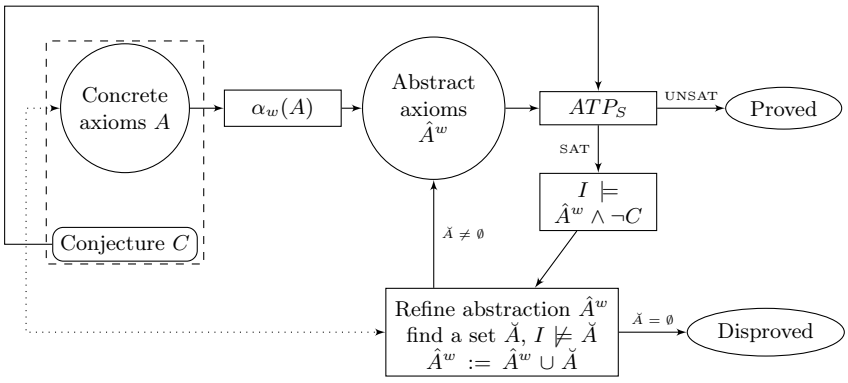

Fig. 4. Under-approximation

4.1 Weakening Abstraction Function

In the case of under-approximation, we propose two weakening abstractions: *instantiation abstraction* and *deletion abstraction*. In the case of instantiation

abstraction, abstraction function generates ground instances of the concrete axioms as it is done in the Inst-Gen framework [19]. In the case of deletion abstraction we delete certain concrete axioms from the theory. This abstraction can be used to incorporate other axioms selection methods into this framework, which are based on removing irrelevant axioms. In particular, we incorporated SInE [15] which selects axioms based on syntactic relevance. In practice, different abstractions can be recombined.

4.2 Strengthening Abstraction Refinement

In the case of deletion abstraction, refinement can be done by adding concrete axioms $\breve{A}$ that turn the model I, which is obtained form ATP_S, into a countermodel, $\breve{A} \subseteq \{\breve{a} \mid \breve{a} \in A, I \not\models \breve{a}\}$. In the case of instantiation abstraction, refinement can be done by generating a set of ground instances of axioms $A\sigma$ such that $I \not\models A\sigma$, $\breve{A} := A\sigma$.

5 Combined approximation

We can combine over- and under-approximations as follows. We use under-approximation in the outer-loop and over-approximation in the place of ATP_S (see Figure 4). Let us note that abstractions can be shared between approximation loops. This combination allows us incorporate other axiom selection methods [15, 30, 32, 33] as part of the under-approximation abstraction and combine them with over-approximation abstractions described in this paper.

6 Evaluation and Experimental Results

We implemented the abstraction-refinement framework described in this paper as part of the current version of iProver v2.7 [18, 19][2], which is also the ATP that we utilised in our experiments.

We evaluated our implementation of the abstraction-refinement framework on the standard benchmark for first-order theorems provers: the TPTP library [29] with the set of problems from the Large Theory Batch (LTB) category in CASC-26 [17, 21, 28], during the competition the wall clock time limit was 90000s per batch. All experiments described in this section were performed using a cluster of computers with the following characteristics: Linux v3.13, cpu 3.1GHz and memory 125GB. We used a time limit of 240s for each attempt to solve a problem.

We experimented with different types of over-approximation abstractions: i) subsumption abstraction, ii) argument filtering abstraction, iii) argument filtering restricted to Skolem functions and splitting predicates, vi) signature grouping abstraction, and v) signature grouping restricted to Skolem functions. We implemented arbitrary combinations of these abstractions, which can be specified as a command line option to iProver, e.g.,

[2] iProver is available at: http://www.cs.man.ac.uk/~ korovink/iprover/

`--abstr_ref "[subs;sig;arg_filter]"`.

For under-approximation abstractions we used the SInE axiom selection algorithm [15] and the Inst-Gen calculus which is the backbone of iProver. SInE is included with Vampire's [20] clausifier, which we also used for clausification.

The first set of experiments were performed over 1500 problems out of which: 716 were solved by signature grouping, 704 by signature grouping of Skolem symbols and constants, 637 by subsumption and 627 by argument filtering. Results are shown in Table 1.

Table 1. Problems solved by over-approximation abstractions with SInE.

Abstraction	**Solutions**
signature grouping	716
signature grouping Skolem/constants	704
subsumption	637
argument filter	627

In the next set of experiments we combined different over-approximation abstractions. In Table 2, we present the results obtained from combining different abstractions. Abstractions were applied in the same order as they are presented.

From these results, we can conclude that combination of abstractions considerably improves the performance. The best combination of abstractions is subsumption, signature grouping and argument filtering which solves around the 55% of the 1500 problems.

Table 2. Problems solved by iProver combination of abstractions and SInE

Abstraction	**Solutions**
subs; sig grouping; arg filter	826
subs; sig grouping	798
subs; sig grouping Skolem/constants; arg filter	733
subs; sig grouping Skolem/constants	719
subs; arg filter	630

We experimented with the top 3 strategies by restricting argument filtering and signature grouping to Skolem functions and splitting predicates and compared these to unrestricted versions. In this experiments the option `--schedule` was set to default. The results are shown in Table 3.

Table 4 shows the number of solutions found by each strategy but excluding the problems solved by the previous ones. The total number of solved problems is 1044. There are several strategies from other combinations of abstractions, which solved small number of problems but turned out that those solutions are unique. If we combine these solutions with solutions shown in Table 4, the total

Table 3. Problems solved by iProver default schedule with abstractions and SInE

Abstraction	**Solutions**
subs; sig grouping; arg filter Skolem/splitting	957
subs; sig grouping; arg filter	942
subs; sig grouping Skolem/constants; arg filter	930

Table 4. Top strategies after removing overlapping solutions with **default schedule**. Where **subs** stands for subsumption, **sig** for signature, **arg-filt** for argument filtering, **SK** restriction to Skolem functions and splitting symbols in iProver.

Abstractions	**Signature**	**Arg-filter**	**Until SAT**	**Solutions**
subs, sig, arg-filt		SK	false	957
subs, sig, arg-filt	SK	default	false	38
subs		default	true	27
subs, sig, arg-filt		default	true	11
subs, sig, arg-filt		default	false	8
subs		default	false	2
subs, sig, arg-filt	SK	default	true	1
			Total	**1044**

number of solutions increases to 1070. Finally, in Table 5 we compare iProver and recent CASC-26 results. From this table we can conclude that integration of combinations of over-approximation abstractions considerably improves performance of iProver. Overall iProver considerably outperforms E-LTB [27] and gets close to the top systems Vampire [20] and MaLARea [32].

Table 5. Comparison with CACS-26 LTB results

Vampire-LTB	MaLARea	iProver-v2.7-all	iProver-v2.7	iProver-LTB-v2.6	E-LTB
1156	1131	1070	957	777	683

7 Conclusion and Further Work

In this paper, we presented a theoretical framework to abstraction-refinement for reasoning with large theories. We presented a number of concrete abstractions based on subsumption, argument filtering and signature grouping and discussed their combinations. We implemented the abstraction-refinement framework in iProver and evaluated different abstractions over the large theory problems in the TPTP library. The results are encouraging and show considerable improvements in the number of overall solved problems in the LTB category. Overall, the number of solved problems is 1070 problems out of 1500 which is considerably larger than the number of problems solved by the previous version of iProver-LTB-2.6 (777) and E-LTB-2.1 (683). Although still below the CASC winner

Vampire-LTB-4.2 (1156) and MaLARea-0.6 (1144). We believe that fine-tuning abstraction parameters will help to further improve the performance.

Acknowledgements. We would like to thank anonymous reviewers for many helpful suggestions.

References

1. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. An extension of lazy abstraction with interpolation for programs with arrays. *Formal Methods in System Design*, 45(1):63–109, 2014.
2. C. Benzmüller, A. Steen, and M. Wisniewski. Leo-iii version 1.1 (system description). In T. Eiter, D. Sands, G. Sutcliffe, and A. Voronkov, editors, *IWIL@LPAR 2017*, volume 1 of *Kalpa Publications in Computing*, pages 11–26. EasyChair, 2017.
3. J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending sledgehammer with SMT solvers. *J. Autom. Reasoning*, 51(1):109–128, 2013.
4. J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Encoding monomorphic and polymorphic types. *Logical Methods in Computer Science*, 12(4), 2016.
5. M. P. Bonacina, C. Lynch, and L. M. de Moura. On deciding satisfiability by theorem proving with speculative inferences. *J. Autom. Reasoning*, 47(2):161–189, 2011.
6. C. E. Brown. Reducing higher-order theorem proving to a sequence of SAT problems. *J. Autom. Reasoning*, 51(1):57–77, 2013.
7. R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding Bit-Vector Arithmetic with Abstraction. In O. Grumberg and M. Huth, editors, *TACAS 2007*, volume 4424 of *LNCS*, pages 358–372, Berlin, Heidelberg, 2007. Springer.
8. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.
9. S. Conchon, A. Goel, S. Krstic, R. Majumdar, and M. Roux. Far-cubicle - A new reachability algorithm for cubicle. In D. Stewart and G. Weissenbacher, editors, *FMCAD 2017*, pages 172–175. IEEE, 2017.
10. T. Gauthier and C. Kaliszyk. Sharing HOL4 and HOL light proof knowledge. In M. Davis, A. Fehnker, A. McIver, and A. Voronkov, editors, *LPAR-20. Proceedings*, volume 9450 of *LNCS*, pages 372–386. Springer, 2015.
11. Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In A. Bouajjani and O. Maler, editors, *CAV 2009. Proceedings*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.
12. B. Glimm, Y. Kazakov, T. Liebig, T. Tran, and V. Vialard. Abstraction refinement for ontology materialization. In M. Bienvenu, M. Ortiz, R. Rosati, and M. Simkus, editors, *Informal Proceedings of the 27th International Workshop on Description Logics*, volume 1193 of *CEUR Workshop Proceedings*, pages 185–196. CEUR-WS.org, 2014.
13. J. C. L. Hernandez and K. Korovin. Towards an abstraction-refinement framework for reasoning with large theories. In T. Eiter, D. Sands, G. Sutcliffe, and A. Voronkov, editors, *IWIL@LPAR 2017*, volume 1 of *Kalpa Publications in Computing*. EasyChair, 2017.
14. K. Hoder, G. Reger, M. Suda, and A. Voronkov. Selecting the selection. In N. Olivetti and A. Tiwari, editors, *IJCAR 2016. Proceedings*, volume 9706 of *LNCS*, pages 313–329. Springer, 2016.

15. K. Hoder and A. Voronkov. Sine Qua Non for Large Theory Reasoning. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE 23. Proceedings*, volume 6803 of *LNCS*, pages 299–314. Springer, 2011.
16. G. Irving, C. Szegedy, A. A. Alemi, N. Eén, F. Chollet, and J. Urban. Deepmath - deep sequence models for premise selection. In D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, editors, *NIPS 2016*, pages 2235–2243, 2016.
17. C. Kaliszyk and J. Urban. HOL(y)Hammer: Online ATP Service for HOL Light. *Mathematics in Computer Science*, 9(1):5–22, 2015.
18. K. Korovin. iProver - an instantiation-based theorem prover for first-order logic (system description). In *IJCAR 2008. Proceedings*, pages 292–298, 2008.
19. K. Korovin. Inst-Gen – A Modular Approach to Instantiation-Based Automated Reasoning. In A. Voronkov and C. Weidenbach, editors, *Programming Logics: Essays in Memory of Harald Ganzinger*, volume 7797 of *LNCS*, pages 239–270. Springer, 2013.
20. L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In N. Sharygina and H. Veith, editors, *Computer Aided Verification - 25th International Conference, CAV. Proceedings*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.
21. R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In *POPL '14*, pages 179–192, 2014.
22. C. Lynch. Unsound Theorem Proving. In J. Marcinkowski and A. Tarlecki, editors, *CSL 2004*, volume 3210 of *LNCS*, pages 473–487. Springer, 2004.
23. D. A. Plaisted. Theorem proving with abstraction. *Artif. Intell.*, 16(1):47–108, 1981.
24. G. Reger, N. Bjørner, M. Suda, and A. Voronkov. AVATAR modulo theories. In C. Benzmüller, G. Sutcliffe, and R. Rojas, editors, *GCAI 2016*, volume 41 of *EPiC Series in Computing*, pages 39–52. EasyChair, 2016.
25. A. Reynolds, C. Tinelli, and L. M. de Moura. Finding conflicting instances of quantified formulas in SMT. In *FMCAD 2014*, pages 195–202. IEEE, 2014.
26. A. Riazanov and A. Voronkov. Splitting without backtracking. In B. Nebel, editor, *IJCAI 2001. Proceedings*, pages 611–617. Morgan Kaufmann, 2001.
27. S. Schulz. System description: E 1.8. In K. L. McMillan, A. Middeldorp, and A. Voronkov, editors, *LPAR-19*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.
28. K. Slind and M. Norrish. A Brief Overview of HOL4. In *TPHOLs 2008. Proceedings*, pages 28–32, 2008.
29. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.
30. G. Sutcliffe and Y. Puzis. SRASS - A Semantic Relevance Axiom Selection System. In *CADE 21*, volume 4603 of *LNCS*, pages 295–310, 2007.
31. A. Teucke and C. Weidenbach. First-order logic theorem proving and model building via approximation and instantiation. In C. Lutz and S. Ranise, editors, *FroCoS 2015. Proceedings*, volume 9322 of *LNCS*, pages 85–100. Springer, 2015.
32. J. Urban. MaLARea: A metasystem for automated reasoning in large theories. *CEUR Workshop Proceedings*, 257:45–58, 2007.
33. J. Urban, G. Sutcliffe, P. Pudlák, and J. Vyskocil. Malarea SG1- machine learner for automated reasoning with semantic guidance. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR 2008. Proceedings*, volume 5195 of *LNCS*, pages 441–456. Springer, 2008.