Implementing a Relevance Tracker Module

Joachim Jansen[1], Jo Devriendt[1], Bart Bogaerts[2,1],
Gerda Janssens[1], Marc Denecker[1]
[1]`first.lastname@kuleuven.be`, [2]`bart.bogaerts@aalto.fi`

[1]KU Leuven, Leuven, Belgium
[2]Aalto University, Espoo, Finland

Abstract. PC(ID) extends propositional logic with inductive definitions: rule sets under the well-founded semantics. Recently, a notion of *relevance* was introduced for this language. This notion determines the set of undecided literals that can still influence the satisfiability of a PC(ID) formula in a given partial assignment. The idea is that the PC(ID) solver can make decisions only on relevant literals without losing soundness and thus safely ignore irrelevant literals.
One important insight that the relevance of a literal is completely determined by the current solver state. During search, the solver state changes have an effect on the relevance of literals. In this paper, we discuss an incremental, lightweight implementation of a relevance tracker module that can be added to and interact with an out-of-the-box SAT(ID) solver.

1 Introduction

Since the addition of conflict-driven clause learning [Marques-Silva and Sakallah, 1999], SAT solvers have made huge leaps forward. Now that these highly-performant SAT-solvers exist, research often stretches *beyond SAT* by extending the language supported by SAT with richer language constructs. Research fields such as SAT Modulo Theories (SMT) [Barrett et al., 2009], Constraint Programming (CP) [Apt, 2003] in the form of lazy clause generation [Stuckey, 2010], or Answer Set Programming (ASP) [Marek and Truszczyński, 1999] could be seen as following this approach. In this paper, we focus on the logic PC(ID): the Propositional Calculus extended with Inductive Definitions [Mariën et al., 2007]. The satisfiability problem for PC(ID) encodings is called SAT(ID) [Mariën et al., 2008]. SAT(ID) can be formalised as SAT modulo a theory of inductive definitions and is closely related to answer set solving. In fact, all the work we introduce in this paper is also applicable to so-called generate-define-test answer set programs.

Recently we [Jansen et al., 2016] defined a notion of *relevance* for SAT(ID). The idea is that we identify a set of literals that can contribute to a solution, and then limit the solver to only make choices on those literals. We call such literals *relevant*. Furthermore, instead of searching for a variable assignment that satisfies the PC(ID) theory, we search for a *partial* assignment that contains sufficient information to guarantee satisfiability. Our approach is based on the notion of *justifications* [Denecker and De Schreye, 1993; Denecker et al., 2015]. As a small example, consider the following theory.

$$
\begin{array}{l}
p_{\mathcal{T}}. \\
\left\{\begin{array}{ll}
p_{\mathcal{T}} & \leftarrow a \wedge b. \\
a & \leftarrow d \vee \neg e \vee f. \\
b & \leftarrow c \vee \neg g \vee h. \\
e & \leftarrow f \vee \neg h \vee i.
\end{array}\right\}
\end{array}
$$

This theory contains one constraint, that $p_{\mathcal{T}}$ must hold, and a definition (between '{' and '}') of $p_{\mathcal{T}}$ in terms of variables a to i. One way to check satisfiability would be to generate an assignment of all variables that satisfies the above theory (this is the classical approach to solving such problems). What we do, on the other hand, is to search for a *partial* assignment to these variables such that $p_{\mathcal{T}}$ is *justified* in that partial assignment. Consider for example the partial assignment where $p_{\mathcal{T}}$, a, b, c and d are true and everything else is unknown. In this assignment, a and b are *justified* because d and c hold respectively; $p_{\mathcal{T}}$ is *justified* because both a and b are justified. This suffices to determine satisfiability of the theory, without considering the definition of e for instance.

Intuitively, a literal is *relevant* if it can contribute to justifying the theory. In the above example, as soon as d is assigned *true*, the variable e becomes *irrelevant*. From that point onwards, search should not take e's defining rule into account.

Jansen et al. [2016] have studied relevance extensively from the theoretical perspective and ran experiments to determine how relevance influences the number of choices and the number of conflicts a solver encounters. These experiments concluded that VSIDS chooses a significant number of relevant literals, and that prevention of decisions on irrelevant literals can lead to significant performance gains in some hand-constructed examples. The actual implementation has not yet been described in detail. That is exactly the goal of this paper. The insights presented in this paper should make it possible to implement relevance in other solvers as well, for instance in ASP solvers. In fact, our implementation heavily relies on components that are already present in modern ASP solvers. To be precise, we reuse propagation mechanisms for inductive definitions and an unfounded set detection algorithm.

The main contributions of this paper are **(1)** the introduction of a method to keep track of justification status in a SAT(ID) solver, **(2)** the introduction of a method to keep track of relevance status in a SAT(ID) solver, and **(3)** a discussion on the properties of these implementations.

2 Preliminiaries: SAT(ID), Relevance

Here we give a short introduction on PC(ID), SAT(ID), justifications, relevance, and recall how relevance can be exploited to improve PC(ID) solvers. For a more elaborate exposition we refer to Jansen et al. [2016].

2.1 PC(ID)

We briefly recall the syntax and semantics of Propositional Calculus extended with Inductive Definitions (PC(ID)) [Mariën, 2009].

A truth value is one of $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$; $\mathbf{t}$ represents *true*, $\mathbf{f}$ *false* and $\mathbf{u}$ *unknown*. The truth order $\leq_t$ on truth values is given by $\mathbf{f} \leq_t \mathbf{u} \leq_t \mathbf{t}$, the precision order $\leq_p$ is given by $\mathbf{u} \leq_p \mathbf{f}$ and $\mathbf{u} \leq_p \mathbf{t}$. Let Σ be a finite set of symbols called *atoms*. A *literal* l is an atom p or its negation $\neg p$. In the former case, we call l *positive*, in the latter, we call l *negative*. We use $\overline{\Sigma}$ to denote the set of all literals over Σ. If l is a literal, we use $|l|$ to denote the atom of l, i.e., to denote p whenever $l = p$ or $l = \neg p$. We use $\sim l$ to denote the literal that is the negation of l, i.e., $\sim p = \neg p$ and $\sim\neg p = p$. Propositional formulas are defined as usual. We use $\varphi \Rightarrow \psi$ for material implication, i.e., as a shorthand for $\neg\varphi \lor \psi$.

A *partial interpretation* $\mathcal{I}$ is a mapping from Σ to truth values. We use the notation $\{p_1^{\mathbf{t}}, \dots, p_n^{\mathbf{t}}, q_1^{\mathbf{f}}, \dots, q_m^{\mathbf{f}}\}$ for the partial interpretation that maps the p_i to $\mathbf{t}$, the q_i to $\mathbf{f}$ and all other atoms to $\mathbf{u}$. We call a partial interpretation *two-valued* if it does not map any atom to $\mathbf{u}$. If $\mathcal{I}$ and $\mathcal{I}'$ are partial interpretations, we say that $\mathcal{I}$ is less precise than $\mathcal{I}'$ (notation $\mathcal{I} \leq_p \mathcal{I}'$) if for all $p \in \Sigma$, $\mathcal{I}(p) \leq_p \mathcal{I}'(p)$. If φ is a propositional formula, we use $\varphi^{\mathcal{I}}$ to denote the truth value ($\mathbf{t}$, $\mathbf{f}$ or $\mathbf{u}$) of φ in $\mathcal{I}$, based on the Kleene truth tables [Kleene, 1938]. If $\mathcal{I}$ is a partial interpretation and l a literal, we use $\mathcal{I}[l : \mathbf{t}]$ to denote the partial interpretation equal to $\mathcal{I}$, except that it interprets l as $\mathbf{t}$ (and similar for $\mathbf{f}$, $\mathbf{u}$). If $\Sigma' \subseteq \Sigma$, we use the notation $\mathcal{I}|_{\Sigma'}$ to indicate the *restriction* of $\mathcal{I}$ to symbols in Σ'. This restriction is a partial interpretation of Σ and satisfies $\mathcal{I}|_{\Sigma'}(p) = \mathbf{u}$ if $p \notin \Sigma'$ and $\mathcal{I}|_{\Sigma'}(p) = \mathcal{I}(p)$ otherwise.

In the rest of this text, when we just say *interpretation*, we mean a two-valued partial interpretation. An interpretation I is completely characterised by the set of atoms p such that $I(p) = \mathbf{t}$. As such, slightly abusing notation, we often identify an interpretation with a set of atoms.

An inductive definition Δ over Σ is a finite set of rules of the form $p \leftarrow \varphi$ where $p \in \Sigma$ and φ is a propositional formula over Σ. We call p the head of the rule and φ the body of the rule. We call p *defined in* Δ if p occurs as the head of a rule in Δ. The set of all symbols defined in Δ is denoted by $\mathit{defs}(\Delta)$. All other symbols are called *open in* Δ. The set of open symbols in Δ is denoted $\mathit{opens}(\Delta)$. We say that a literal l is *defined in* Δ if $|l| \in \mathit{defs}(\Delta)$. We use the *parametrised well-founded semantics* for inductive definitions [Denecker and Vennekens, 2007]. That is, interpretation I is a model of Δ (denoted $I \models \Delta$) if I is the well-founded model of Δ in context $I|_{\mathit{opens}(\Delta)}$. We call an inductive definition *total* if for every interpretation I of the open symbols, the well-founded model in context I is a two-valued interpretation.

A *PC(ID)* theory $\mathcal{T}$ over Σ is a set of propositional formulas, called constraints, and inductive definitions over Σ. Interpretation I is a model of $\mathcal{T}$ if I is a model of all definitions and constraints in $\mathcal{T}$. Without loss of generality [Mariën, 2009], we assume that every PC(ID) theory is in **Definition Normal Form (DEFNF)**, where $\mathcal{T} = \{p_{\mathcal{T}}, \Delta\}$ and

- $p_{\mathcal{T}}$ is an atom,

- Δ is an inductive definition defining $p_{\mathcal{T}}$,
- every rule in Δ is of the form $p \leftarrow l_1 \odot \cdots \odot l_n$, where $\odot$ is either $\wedge$ or $\vee$, p is an atom, and each of the l_i are literals,
- every atom p is defined in at most one rule of Δ.

A rule in which $\odot$ is $\wedge$, respectively $\vee$ is called a *conjunctive*, respectively *disjunctive*, rule. The rules in a definition Δ impose a *direct dependency relation*, denoted dd_Δ, between literals, defined as follows. For every rule $p \leftarrow l_1 \odot \cdots \odot l_n$ in Δ, dd_Δ contains (p, l_i) and $({\sim}p, {\sim}l_i)$ for all $1 \leq i \leq n$. The *dependency graph* of Δ is the graph $G_\Delta = (\overline{\Sigma}, dd_\Delta)$.

For the remainder of the paper, we assume that some PC(ID) theory $\mathcal{T} = \{p_{\mathcal{T}}, \Delta\}$ is fixed.

It has been argued many times before [Denecker, 1998; Denecker and Ternovska, 2008; Denecker and Vennekens, 2014] that all sensible definitions in mathematical texts are total definitions. Following these arguments, in the rest of this paper we assume Δ to be a total definition.

The satisfiability problem for PC(ID), i.e., deciding whether a PC(ID) theory has a model, is called *SAT(ID)*. This problem is NP-complete [Mariën et al., 2008].

2.2 Justifications

Consider a directed graph $G = (V, E)$, with V a set of nodes and E a set of edges. If G contains an edge from l to l' (i.e., $(l, l') \in E$), we say that l is a parent of l' in G and that l' is a child of l in G. A node l is called a *leaf* of G if it has no children in G; otherwise it is called *internal* in G. Let $G' = (V', E')$ be another graph. We define the union of two graphs (denoted $G \cup G'$) as the graph with vertices $V \cup V'$ and edges $E \cup E'$.

Suppose l is a literal with $p = |l|$ and $p \in \mathit{defs}(\Delta)$ with defining rule $p \leftarrow l_1 \odot \cdots \odot l_n$. A set of literals J_d is a *direct justification* of l in Δ if one of the following holds:

- $l = p$, $\odot$ is $\wedge$, and $J_d = \{l_1, \ldots, l_n\}$,
- $l = p$, $\odot$ is $\vee$, and $J_d = \{l_i\}$ for some i,
- $l = \neg p$, $\odot$ is $\wedge$, and $J_d = \{{\sim}l_i\}$ for some i,
- $l = \neg p$, $\odot$ is $\vee$, and $J_d = \{{\sim}l_1, \ldots, {\sim}l_n\}$.

Note that a direct justification of a literal can only contain children of that literal in the dependency graph.

A *justification* [Denecker and De Schreye, 1993; Denecker et al., 2015] J of a definition Δ is a subgraph of G_Δ, such that each internal node $l \in J$ is a defined literal and the set of its children is a direct justification of l in Δ. We say that J *contains* l if l occurs as node in J. A justification is *total* if none of its leaves are defined literals. A justification can contain *cycles*.[1] A cycle is called *positive*

[1] In this text, we assume that Δ is finite; in this case cycles are simply loops in the graph. The infinite case is a bit more subtle, and an adapted definition of cycle is required to maintain all results presented below.

(resp. *negative*) if it contains only positive (resp. negative) literals. It is called a *mixed* cycle otherwise.

If J is a justification and $\mathcal{I}$ a (partial) interpretation, we define the value of J in $\mathcal{I}$, denoted $V_\mathcal{I}(J)$ as follows:

- $V_\mathcal{I}(J) = \mathbf{f}$ if J contains a leaf l with $l^\mathcal{I} = \mathbf{f}$ or a positive cycle (or both).
- $V_\mathcal{I}(J) = \mathbf{u}$ if $V_\mathcal{I}(J) \neq \mathbf{f}$ and J contains a leaf l with $l^\mathcal{I} = \mathbf{u}$ or a mixed cycle (or both).
- $V_\mathcal{I}(J) = \mathbf{t}$ otherwise (all leaves are $\mathbf{t}$ and cycles, if any, are negative).

A literal l is *justified* (in $\mathcal{I}$, for $\mathcal{T}$) if there exists a total justification J (of Δ) that contains l such that $V_\mathcal{I}(J) = \mathbf{t}$. In this case, we say that such a J *justifies* l (in $\mathcal{I}$, for $\mathcal{T}$). It follows from the definition that it is not possible that both l and $\neg l$ are justified in the same in the same interpretation.

The *justification status* of an atom p (in $\mathcal{I}$, for $\mathcal{T}$) is defined as follows. The justification status of p is *true* if and only if the literal p is justified in $\mathcal{I}$ for $\mathcal{T}$. The justification status of p is *false* if and only if the literal $\neg p$ is justified in $\mathcal{I}$ for $\mathcal{T}$. Otherwise the justification status of p is *unknown*.

Denecker and De Schreye [1993] and later also Denecker et al. [2015] showed that many semantics of logic programs can be captured by justifications.

2.3 Relevance

Now, we recall the central definitions and theorems related to relevance. For a more detailed exposition of the theoretical foundations, we refer to Jansen et al. [2016].

Theorem 1 ([Jansen et al., 2016]; Theorem 3.1). *$\mathcal{T}$ is satisfiable if and only if there exists a partial interpretation $\mathcal{I}$ and a justification J that justifies $p_\mathcal{T}$ in $\mathcal{I}$.*

Definition 1 ([Jansen et al., 2016]; Definition 3.2). *Given a PC(ID) theory $\mathcal{T} = \{p_\mathcal{T}, \Delta\}$ and a partial interpretation $\mathcal{I}$, we inductively define the set of relevant literals, denoted $\mathcal{R}_{\mathcal{T},\mathcal{I}}$, as follows*

- *$p_\mathcal{T}$ is relevant if $p_\mathcal{T}$ is not justified,*
- *l is relevant if l is not justified and there exists some l' such that $(l', l) \in dd_\Delta$ and l' is relevant.*

The central theorem regarding relevance shows that any search algorithm that arrives in a state in which $p_\mathcal{T}$ is justified by deciding on a literal l that is irrelevant can also arrive in such a state without deciding on l. Hence, if a literal l is irrelevant, choosing on l does not help justifying $p_\mathcal{T}$.

Theorem 2 ([Jansen et al., 2016]; Theorem 3.5). *Let $\mathcal{T} = \{p_\mathcal{T}, \Delta\}$ be a PC(ID) theory. Suppose $\mathcal{I}$ is a partial interpretation and l a literal such that $\mathcal{I}(|l|) = \mathbf{u}$ and l is not relevant in $\mathcal{I}$. If $p_\mathcal{T}$ is justified in some partial interpretation $\mathcal{I}'$ more precise than $\mathcal{I}$, then $p_\mathcal{T}$ is also justified in $\mathcal{I}'[l : \mathbf{f}]$ and in $\mathcal{I}'[l : \mathbf{t}]$.*

Consequently, the work suggests adapting SAT(ID) solvers such that they **(1)** do not make decisions on irrelevant literals, and **(2)** stop searching when there are no relevant literals left. This requires the underlying solver to keep track of which literals are relevant. This task is incremental in nature: small changes to the state of the solver will result in small changes to the relevance of literals. Since modern solvers work with hunderds of thousands of variables and go through millions of assignment in quick succession, it is essential to do these changes as efficiently as possible.

3 The Basic Framework

The aim of this work is to discuss the implementation of an algorithm to keep track of relevant literals.

As said in Theorem 1, the solver aims to arrive at a state $\mathcal{S}$ where $p_{\mathcal{T}}$ is justified in $\mathcal{I}$. The solver does this by making decisions, performing propagation, and backtracking. To prevent the solver from making "useless" decisions, we need to know whether literals are relevant or not in $\mathcal{I}$.

We consider the underlying solver to have an internal state $\mathcal{S}$ of the form $\mathcal{S} = \langle \overline{\Sigma}, \mathcal{T}, \mathcal{I} \rangle$, with **(1)** $\overline{\Sigma}$ denoting the set of literals used in the solver, **(2)** $\mathcal{T} = \{p_{\mathcal{T}}, \Delta\}$ a DEFNF theory, and **(3)** $\mathcal{I}$ the current partial interpretation in the solver.

During the search process, the (CDCL) solver adds learned conflict clauses to the theory. However, learned conflict clauses are logical consequences of the theory and because of this we do not consider them to be a part of the theory $\mathcal{T}$ in $\mathcal{S}$. Instead, $\mathcal{T}$ is reserved for non-learned clauses. We assume $\mathcal{T}$ to remain static during the search process. This assumption is valid in most *ground-and-solve* systems. Recent work focuses on interleaving this process [De Cat et al., 2015]. Extending this work to allow for a changing theory is future work, but should be of limited complexity given the framework we present here.

The relevance tracker needs to take into account changes in the solver state, in particular in $\mathcal{I}$. Before we define the interface between the relevance tracker and the solver, we discuss the solver state and its changes.

During the search process, the solver iteratively performs one of the following state changes:

- $\langle \overline{\Sigma}, \mathcal{T}, \mathcal{I} \rangle \mapsto \langle \overline{\Sigma}, \mathcal{T}, \mathcal{I}[l : \mathbf{t}] \rangle$ a literal l becomes *true*, or
- $\langle \overline{\Sigma}, \mathcal{T}, \mathcal{I} \rangle \mapsto \langle \overline{\Sigma}, \mathcal{T}, \mathcal{I}[l : \mathbf{u}] \rangle$ a literal l becomes *unknown*.

Note that this set of operations allows the solver to make a literal l *false* by making literal $\sim l$ *true*.

In order to get the necessary information about the changes of the solver and to implement the above revision problem, the relevance tracker listens to notifications. The relevance tracker supports the following interface to the underlying solver.

notifyBecomesTrue a literal l becomes *true* in $\mathcal{I}$

notifyBecomesUnknown a literal l becomes *unknown* in $\mathcal{I}$
isRelevant query whether a given literal l is relevant (returns a boolean value)

Methods **notifyBecomesTrue** and **notifyBecomesUnknown** must be called by the underlying solver when a literal has become true, respectively unknown. The **isRelevant** method is used by the solver to ask the module whether the given literal is relevant. The relevance information allows the solver to change its underlying heuristic, selecting only relevant literals.

4 Deriving the justification status of literals

The definition of relevance relies on knowledge about which literals are *justified* in the solver. In this section, we discuss how we implemented extraction of this information. We opted to implement a method that re-uses the underlying SAT(ID) solver to keep track of the justification status of literals. The method creates a new atom, called the "justification atom", for each defined atom p, denoted as $j(p)$. We call a literal $j(p)$ or $\neg j(p)$ a *justification literal.*

The intended interpretation of $j(p)$ is that $j(p)$ is true iff p is justified, $j(p)$ is false iff $\neg p$ is justified and $j(p)$ is unknown otherwise. To ensure that justification literals indeed get the right value, an extra PC(ID) definition Δ_j, denoted the "justification definition", is added to the theory $\mathcal{T}$. Δ_j is constructed based on the original definition Δ in the following manner. The existing definition Δ is copied, except that every defined atom p is replaced with the newly created atom $j(p)$. Thus, of all the atoms in the original definition, only the open atoms remain.

Example 1. Transforming the original definition

$$\Delta = \left\{ \begin{array}{rl} p_{\mathcal{T}} \leftarrow & c_1 \wedge c_2 \wedge c_3 \wedge c_4 \\ c_1 \leftarrow & \neg b \vee \neg d \\ c_2 \leftarrow & a \vee b \vee \neg c \\ c_3 \leftarrow & \neg b \vee e \vee \neg f \\ c_4 \leftarrow & d \vee f \vee \neg a \\ f \leftarrow & b \vee d \end{array} \right\}$$

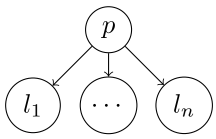
leads to the justification definition

$$\Delta_j = \left\{ \begin{array}{rl} j(p_{\mathcal{T}}) \leftarrow & j(c_1) \wedge j(c_2) \wedge j(c_3) \wedge j(c_4) \\ j(c_1) \leftarrow & \neg b \vee \neg d \\ j(c_2) \leftarrow & a \vee b \vee \neg c \\ j(c_3) \leftarrow & \neg b \vee e \vee \neg j(f) \\ j(c_4) \leftarrow & d \vee j(f) \vee \neg a \\ j(f) \leftarrow & b \vee d \end{array} \right\}$$

In addition to the creation of this new definition Δ_j, we prohibit the solver from making choices on these justification atoms. Because of this, the value of all $j(p)$ will be purely the result of the underlying propagation mechanism for definitions. Our claim is now that existing propagation mechanisms will propagate *exactly* those literals that are justified. We assume a solver that performs *unit propagation* and *unfounded set propagation* [Gebser et al., 2012; Mariën et al., 2007], i.e., propagation that makes all atoms in an unfounded set false.

Theorem 3. *Let Δ be a (total) definition and $\mathcal{I}$ a partial interpretation in which all defined symbols of Δ are interpreted as* $\mathbf{u}$. *Let l be a defined literal in Δ. In this case l is justified in $\mathcal{I}$ if and only if l is derivable by unit propagation on the completion*[2] *[Clark, 1978] of Δ and unfounded set propagations.*

Proof (Sketch of the proof). Intuitively, from a sequence of propagations, we can create a justification and vice versa: each justification induces a sequence of propagations. The correspondence is as follows. First for the completion, if Δ contains a rule $p \leftarrow l_1 \wedge \cdots \wedge l_n$, then this rule propagates $p = \mathbf{t}$ if and only if each of the l_i is true. This corresponds to the justification



And similar justification constructs can be defined for when $\neg p$ is propagated or when the rule is disjunctive.

Unfounded set propagation essentially corresponds to a justification of a set of negative facts by a negative cycle.

The condition that a justification can have no mixed or positive cycles corresponds to the fact that propagation must happen in order. E.g., from the rule $p \leftarrow p \vee q$, p can only be propagated if q is true; p cannot be propagated "because p is true". $\square$

The previous theorem establishes that our approach works; a justification literal $j(p)$ will be propagated to true if p is justified (note that p is justified iff $j(p)$ is justified). It also explains why we use a duplicated definition: the theorem only holds if $\mathcal{I}$ is an $opens(\Delta)$ interpretation. Since this cannot be enforced (we don't want to intrude in the solver's search), we make a copy and never make choices on the copied defined symbols.

Thus, we extend our solver state $\mathcal{S} = \langle \overline{\Sigma}, \mathcal{T}, \mathcal{I} \rangle$ to a $\mathcal{S}' = \langle \overline{\Sigma}', \mathcal{T}', \mathcal{I}', \Sigma' \rangle$ with

- Σ' = set containing the newly introduced justification atoms that the solver cannot decide on
- $\overline{\Sigma}' = \overline{\Sigma} \cup \overline{\Sigma'}$
- $\mathcal{T}' = \{p_{\mathcal{T}}, \Delta'\}$ if $\mathcal{T} = \{p_{\mathcal{T}}, \Delta\}$ and $\Delta' = \Delta \cup \Delta_j$
- $\mathcal{I}'$ = a partial structure over $\overline{\Sigma}'$

With all this in place, we derive the interpretation for $justified(l)$ as follows.

- $justified(p)$ is true if and only if $j(p)$ is true in $\mathcal{I}'$, and
- $justified(\neg p)$ is true if and only if $j(p)$ is false in $\mathcal{I}'$.

[2] The completion of a rule $p \leftarrow q$ is the underapproximation using the first-order logic (FO) sentence $p \Leftrightarrow q$, which demands that p and q hold equal truth values

5 Implementing the relevance tracker

The source code that implements the techniques discussed here can be found at `https://dtai.cs.kuleuven.be/static/krr/files/experiments/idp_relevance_experiments.tar.gz` .

The solver maintains a subgraph of the dependency graph in order to keep track of the set of relevant literals. This subgraph, denoted as the *relevance graph* contains all literals that are relevant and all edges between them (in the dependency graph). As such, the task of the tracker is to determine whether a given literal is a member of this graph or not. We store this graph using a data structure, denoted $\texttt{candidate\_parents(}l\texttt{)}$ that associates a literal with a set of literals called "candidate parents". The "candidate parents" of l are the literals that are parents of l in the relevance graph. I.e., if l is irrelevant, this set is empty, otherwise, it consists of all relevant parents of l in the dependency graph. As such, it can be seen that there is an edge (p, l) in the relevance graph iff $p \in \texttt{candidate\_parents(}l\texttt{)}$. Thus, l is relevant if and only if l has a non-empty set of candidate parents. We now describe an incremental algorithm to update the set of candidate parents for all literals if the state of the solver changes. We prefer to keep these changes *local*, i.e., to not reconstruct the entire relevance graph with each solver change.

When the solver state changes, and the set of candidate parents must be updated, care must be taken to detect and remove cyclic dependencies. These cyclic dependencies can arise when a candidate parent is removed from a literal l and the remaining candidate parents of that literal are not reachable from $p_\mathcal{T}$ anymore but still have l as a candidate parent, creating a loop. A more detailed example is given in Example 2

Example 2. The following definition has the cyclic dependency of $p \leftarrow q \leftarrow p$.

$$\left\{ \begin{array}{c} p_\mathcal{T} \leftarrow a \lor p \\ p \leftarrow q \\ q \leftarrow p \end{array} \right\}$$

Initially $\mathcal{I} = \emptyset$, thus nothing is justified and all literals are relevant. Thus, p has candidate parents $\{p_\mathcal{T}, q\}$, and q has candidate parents $\{p\}$.

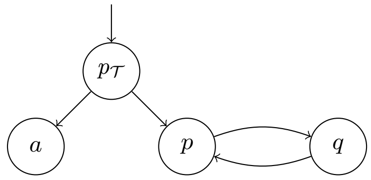


Fig. 1. Relevance graph for $\mathcal{I} = \{\}$

Consider the case where a becomes true and hence $p_\mathcal{T}$ becomes justified.

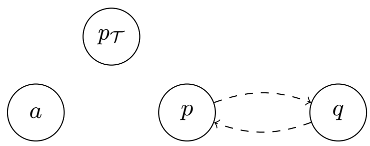


Fig. 2. Relevance graph for $\mathcal{I} = \{a^{\mathbf{t}}\}$. Remaining loop indicated with dashed edges.

Simply removing $p_{\mathcal{T}}$ from the set of candidate parents of p means that p still has candidate parents $\{q\}$, which is actually just a loop leading back to p. Thus, a cycle detection algorithm is needed to force p and q to remain loop-free.

Thus, adding and removing candidate parents is a complicated matter. In Section 5.4 we discuss how this cycle detection is done. For now, we use the following interface for adjusting the set of a candidate parents.

notifyAddCandidateParent(l,l') add l' to the candidate parents of l
notifyRemoveCandidateParent(l,l') remove l' from the candidate parents of l

Our definition of candidate parents potentially changes when the following changes take place (note that we already assumed the dependency relation to be non-changeable).

- A change in the justification status of l
- A change in the relevance status of a parent literal l'

Thus, we extend the interface of the relevance tracker to also support the following methods.

notifyBecomesJustified(l) A literal l goes from unjustified to justified
notifyBecomesUnjustified(l) A literal l goes from justified to unjustified
notifyBecomesRelevant(l) A literal l goes from irrelevant to relevant
notifyBecomesIrrelevant(l) A literal l goes from relevant to irrelevant

In the following subsections we present **(1)** an overview of the data structures in the relevance tracker and **(2)** the algorithms for the methods in our interface.

5.1 Data Structures

The data structures include sets and maps. Unless specified otherwise, we use hash sets and hash maps. The implementation uses `std::unordered_set` and `std::unordered_map` provided by the `C++` standard library.

Internally, we store the dependency relation dd_{Δ} using two maps in our module, named `children` and `parents`. These data structures map a literal to a set of literals. The first map (`children`) maps a literal to its set of children in dd_{Δ}. The second map (`parents`) maps a literal to its set of parents in dd_{Δ}. They are initialised using the **notifyNewRule** method. Once they are initialised, they remain fixed.

We use a map (`to_just_lit`) to transform a normal literal to its justification literal ($p \mapsto j(p)$, $\neg p \mapsto \neg j(p)$). For efficiency reasons, we also maintain the inverse map $\texttt{to\_nonjust\_lit} = \texttt{to\_just\_lit}^{-1}$. These maps are initialised when the justification definition Δ_j is created and do not change during execution afterwards.

We maintain a set of atoms (`is_just_atom`) to identify the justification atoms that were introduced. This set are initialised when the justification definition Δ_j is created and does not change during execution afterwards.

We use round brackets to indicate the result of a map lookup, e.g.,

$$\texttt{to\_just\_lit}(p) \texttt{ = } j(p).$$

We use round brackets to do a containment check of sets. More precisely,

$$\texttt{is\_just\_atom}(p) = \texttt{true}$$

if and only if p is in the set `is_just_atom`. As mentioned before, the underlying solver is not allowed to make decisions on literals in this set.

We maintain a map `candidate_parents` with the invariant that it maps a literal l to the set of candidate parents of l. This map is dynamic throughout execution and changes to this map are performed using the **notifyAddCandidateParent** and **notifyRemoveCandidateParent** methods.

5.2 Notification-based Algorithms

Given that the invariant of `candidate_parents` is satisfied in solver state $\mathcal{S} = \langle \overline{\Sigma}, \mathcal{T}, \mathcal{I} \rangle$, we wish to perform the necessary changes such that they are satisfied in solver state $\mathcal{S}' = \langle \overline{\Sigma}, \mathcal{T}, \mathcal{I}[p : tv] \rangle$ with p some atom and tv one of $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$.

We initiate our notification-based algorithm as follows. If $tv = \mathbf{t}$, then we call **notifyBecomesTrue(**p**)**. If $tv = \mathbf{f}$, then we call **notifyBecomesTrue(**$\neg p$**)**. If $tv = \mathbf{u}$, then we call **notifyBecomesUnknown(**p**)**.

This call, in turn, can trigger other internal notifications. The implementation of these cascading notifications ensures that `candidate_parents` will comply with its invariant in interpretation $\mathcal{S}'$ after the designated call to **notifyBecomesTrue** or **notifyBecomesUnknown** is complete.

The relevance tracker implements **isRelevant(**l**)** by checking whether `candidate_parents`(l) maps to an empty set or not. This is a correct representation of the relevance status of l if the invariant of `candidate_parents` is satisfied.

For methods **notifyBecomesTrue(**l**), notifyBecomesUnknown(**l**)**: the given literal can be a normal literal (p or $\neg p$) or a justification literal ($j(p)$ or $\neg j(p)$). The relevance tracker takes no action for normal literals. If the given literal is a justification literal, then we retrieve the original normal literal and notify the relevance tracker that this literal has become (un)justified. Note that we re-use the notation of $|l|$ to indicate the atom of literal l.

notifyBecomesTrue(l**)**: if `is_just_atom`($|l|$), then call **notifyBecomesJustified(**`to_nonjust_lit`(l)**)**.

notifyBecomesUnknown(l): if `is_just_atom`($|l|$), then call **notifyBecomesUnjustified(**`to_nonjust_lit(`l`)`**)**.
notifyBecomesJustified(l): call **notifyRemoveAllCandidateParentsOf(l)**.
notifyBecomesUnjustified(l): for all parents p of l that are relevant, call **notifyAddCandidateParent(l,p)**.
notifyBecomesRelevant(l): for all children c of l, call **notifyAddCandidateParent(c,l)**.
notifyBecomesIrrelevant(l): for all children c of l, call **notifyRemoveCandidateParent(c,l)**.

5.3 Maintaining watches instead of sets of candidates

The above methods dictate how the `candidate_parents` map should be manipulated. For efficiency reasons, the relevance tracker does not actively maintain this set of candidate parents. Instead it keeps track of a single candidate parent as "watched" parent. This watched parent is maintained using a map called `watched_parent(`l`)` that maps a literal to a single parent of l. The method **isRelevant(l)** now checks whether a given literal l has a watched parent or not.

We only keep track of a single watched parent in order to minimize how many times a cycle detection algorithm has to be invoked. The manipulation of the set of candidate parents, along with the invocation of a cycle detection algorithm is done as follows

notifyAddCandidateParent(l,l') Check for the following criteria
- l does not have a watched parent yet
- l is not justified
- l' is relevant
- l is a child of l'

If they are met, make `watched_parent(`l`) = `l' and call **notifyBecomesRelevant(l)**. Note that a cyclic dependency check between l and l' is not needed, since l could not have been a suitable watch for any other literal, as it was not relevant before.

notifyRemoveCandidateParent(l,l') If l had l' as its watch, remove l' as watched parent of l. Try to find an alternative candidate parent n such that the following hold.
- $n \neq l'$
- l is a child of n
- l is not justified
- n is relevant
- Use a cycle detection algorithm to verify that setting `watched_parent(`l`) = `n would not create a dependency cycle

If such n can be found, set `watched_parent(`l`) = `n. If such n cannot be found, call **notifyBecomesIrrelevant(l)**.

The implementation for the search for an alternative watch is a re-use of an existing "unfounded set detection" algorithm. This algorithm is considered the fastest algorithm to achieve this task to date.

5.4 Detecting Cycles

For our implementation of the detection of cycles, we re-use parts of the existing *unfounded set propagation* algorithm [Gebser et al., 2012; Mariën et al., 2007]. This algorithm has a subcomponent that searches for cycles over negative literals.

6 Conclusion and Future Work

The notion of relevance is related to Magic Sets [Bancilhon et al., 1986; Beeri and Ramakrishnan, 1991] in the field of Logic Programming in the sense that the resulting program of the magic set transformation on a program P and query Q does not execute parts of P that do not contribute towards solving query Q.

One area where relevance should give great speedups is stable model counting. When $p_{\mathcal{T}}$ is justified, the number of solutions that this partial assignment represents is equal to 2^n with n the number of unassigned open atoms in Δ. Stable model counters generally stop when the *justified residual program* [Aziz et al., 2015] is empty. Whenever this occurs, $p_{\mathcal{T}}$ is justified. However, this does not hold the other way round. There are other cases where $p_{\mathcal{T}}$ is justified, but the justified residual program is non-empty. As such, exploiting relevance can ensure cutting out bigger parts of the search tree when model counting.

In this paper we have presented our implementation of a relevance tracker module on top of an existing SAT(ID) solver, which consists of two methods; one to keep track of the set of justified literals and one to keep track of the set of relevant literals. Each of them reuses existing techniques, namely the following.

- Our method for keeping track of the justification status of literals. This method reuses existing propagation mechanisms. We also prove the correctness of this approach if the underlying solver guarantees completeness with respect to rule application and unfounded set propagation.
- Our method for keeping track of the relevance status of literals reuses the existing unfounded set detection techniques [Gebser et al., 2012; Mariën et al., 2007] for detection of cyclic dependencies between "candidate parents".

Generate-and-test ASP programs are the most common form of ASP programs, as can be witnessed, e.g., from the benchmarks in the latest ASP competitions [Calimeri et al., 2016]. Generate-and-test ASP programs closely correspond to PC(ID) theories [Denecker et al., 2012]. This paper imposes minimal assumptions on the underlying solver, thus making it possible to translate these ideas to an ASP context. Since experiments have shown [Jansen et al., 2016] that exploiting relevance during solving can reduce the number of decisions, as well as the number of conflicts, it would be interesting to see how relevance might possibly affects existing ASP solvers.

Acknowledgements

This research was supported by the project GOA 13/010 Research Fund KU Leuven and projects G.0489.10, G.0357.12 and G.0922.13 of FWO (Research Foundation - Flanders). Bart Bogaerts is supported by the Finnish Center of Excellence in Computational Inference Research (COIN) funded by the Academy of Finland (grant #251170).

References

Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

Rehan Abdul Aziz, Geoffrey Chu, Christian J. Muise, and Peter James Stuckey. Stable model counting and its application in probabilistic logic programming. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 3468–3474. AAAI Press, 2015. ISBN 978-1-57735-698-1. URL `http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9709`.

Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM. ISBN 0-89791-179-2. doi: 10.1145/6012.15399. URL `http://doi.acm.org/10.1145/6012.15399`.

Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009. ISBN 978-1-58603-929-5.

Catriel Beeri and Raghu Ramakrishnan. Special issue: Database logic progamming on the power of magic. *The Journal of Logic Programming*, 10(3):255 – 299, 1991. ISSN 0743-1066. doi: http://dx.doi.org/10.1016/0743-1066(91)90038-Q. URL `http://www.sciencedirect.com/science/article/pii/074310669190038Q`.

Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. Design and results of the fifth answer set programming competition. *Artif. Intell.*, 231:151–181, 2016. doi: 10.1016/j.artint.2015.09.008. URL `http://dx.doi.org/10.1016/j.artint.2015.09.008`.

Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322. Plenum Press, 1978. ISBN 0-306-40060-X.

Broes De Cat, Marc Denecker, Maurice Bruynooghe, and Peter J. Stuckey. Lazy model expansion: Interleaving grounding with search. *J. Artif. Intell. Res. (JAIR)*, 52:235–286, 2015. doi: 10.1613/jair.4591. URL `http://dx.doi.org/10.1613/jair.4591`.

Marc Denecker. The well-founded semantics is the principle of inductive definition. In Jürgen Dix, Luis Fariñas del Cerro, and Ulrich Furbach, editors, *JELIA*, volume 1489 of *LNCS*, pages 1–16. Springer, 1998. ISBN 3-540-65141-1.

Marc Denecker and Danny De Schreye. Justification semantics: A unifying framework for the semantics of logic programs. In Luís Moniz Pereira and Anil Nerode, editors, *LPNMR*, pages 365–379. MIT Press, 1993. ISBN 0-262-66083-0. URL `https://lirias.kuleuven.be/handle/123456789/133075`.

Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log.*, 9(2):14:1–14:52, April 2008. ISSN 1529-3785. URL http://dx.doi.org/10.1145/1342991.1342998.

Marc Denecker and Joost Vennekens. Well-founded semantics and the algebraic theory of non-monotone inductive definitions. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *LPNMR*, volume 4483 of *Lecture Notes in Computer Science*, pages 84–96. Springer, 2007. ISBN 978-3-540-72199-4. doi: 10.1007/978-3-540-72200-7_9. URL http://dx.doi.org/10.1007/978-3-540-72200-7_9.

Marc Denecker and Joost Vennekens. The well-founded semantics is the principle of inductive definition, revisited. In Chitta Baral, Giuseppe De Giacomo, and Thomas Eiter, editors, *KR*, pages 1–10. AAAI Press, 2014. ISBN 978-1-57735-657-8. URL http://www.aaai.org/ocs/index.php/KR/KR14/paper/view/7957.

Marc Denecker, Yuliya Lierler, Mirosław Truszczyński, and Joost Vennekens. A Tarskian informal semantics for answer set programming. In Agostino Dovier and Vítor Santos Costa, editors, *ICLP (Technical Communications)*, volume 17 of *LIPIcs*, pages 277–289. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012. ISBN 978-3-939897-43-9.

Marc Denecker, Gerhard Brewka, and Hannes Strass. A formal theory of justifications. In Francesco Calimeri, Giovambattista Ianni, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, volume 9345 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2015. ISBN 978-3-319-23263-8. doi: 10.1007/978-3-319-23264-5_22. URL http://dx.doi.org/10.1007/978-3-319-23264-5_22.

Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012.

Joachim Jansen, Bart Bogaerts, Jo Devriendt, Gerda Janssens, and Marc Denecker. Relevance for SAT(ID). In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 596–603. IJCAI/AAAI Press, 2016. ISBN 978-1-57735-770-4. URL http://www.ijcai.org/Abstract/16/091.

Stephen Cole Kleene. On notation for ordinal numbers. *The Journal of Symbolic Logic*, 3(4):150–155, 1938. ISSN 00224812. URL http://www.jstor.org/stable/2267778.

Victor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In Krzysztof R. Apt, Victor Marek, Mirosław Truszczyński, and David S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999. URL http://arxiv.org/abs/cs.LO/9809032.

Maarten Mariën. *Model Generation for ID-Logic*. PhD thesis, Department of Computer Science, KU Leuven, Belgium, February 2009.

Maarten Mariën, Johan Wittocx, and Marc Denecker. Integrating inductive definitions in SAT. In Nachum Derschowitz and Andrei Voronkov, editors, *LPAR*, volume 4790 of *LNCS*, pages 378–392. Springer, 2007. ISBN 3-540-75558-6.

Maarten Mariën, Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In Hans Kleine Büning and Xishun Zhao, editors, *SAT*, volume 4996 of *LNCS*, pages 211–224. Springer, 2008. ISBN 978-3-540-79718-0.

João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

Peter J. Stuckey. Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving. In *CPAIOR*, pages 5–9, 2010.