Journal of Artificial Intelligence Research 52 (2015) 235-286 Submitted 10/14; published 02/15

Lazy Model Expansion: Interleaving Grounding with Search

Broes De Cat BROES.DECAT@GMAIL.COM
OM Partners, Belgium

Marc Denecker MARC.DENECKER@CS.KULEUVEN.BE
Dept. Computer Science, KULeuven, Belgium

Peter Stuckey PSTUCKEY@UNIMELB.EDU.AU
National ICT Australia and
Dept. of Computing and Information Systems
The University of Melbourne, Australia

Maurice Bruynooghe MAURICE.BRUYNOOGHE@CS.KULEUVEN.BE
Dept. Computer Science, KULeuven, Belgium

Abstract

Finding satisfying assignments for the variables involved in a set of constraints can be cast as a (bounded) *model generation* problem: search for (bounded) models of a theory in some logic. The state-of-the-art approach for bounded model generation for rich knowledge representation languages like Answer Set Programming (ASP) and FO($\cdot$) and a CSP modeling language such as Zinc, is *ground-and-solve*: reduce the theory to a ground or propositional one and apply a search algorithm to the resulting theory.

An important bottleneck is the blow-up of the size of the theory caused by the grounding phase. *Lazily grounding* the theory during search is a way to overcome this bottleneck. We present a theoretical framework and an implementation in the context of the FO($\cdot$) knowledge representation language. Instead of grounding all parts of a theory, *justifications* are derived for some parts of it. Given a partial assignment for the grounded part of the theory and valid justifications for the formulas of the non-grounded part, the justifications provide a recipe to construct a complete assignment that satisfies the non-grounded part. When a justification for a particular formula becomes invalid during search, a new one is derived; if that fails, the formula is split in a part to be grounded and a part that can be justified. Experimental results illustrate the power and generality of this approach.

1. Introduction

The world is filled with combinatorial problems. These include important combinatorial optimization tasks such as planning, scheduling and rostering, combinatorics problems such as extremal graph theory, and countless puzzles and games. Solving combinatorial problems is hard, and all the methods we know to tackle them involve some kind of search.

Various *declarative paradigms* have been developed to solve such problems. In such approaches, objects and attributes that are searched for are represented by symbols, and constraints to be satisfied by those objects are represented as expressions over these symbols in a declarative language. Solvers then search for values for these symbols that satisfy the constraints. This idea is found in the fields of Constraint Programming (CP) (Apt, 2003), ASP (Marek & Truszczyński, 1999), SAT, Mixed Integer Programming (MIP), etc. In the terminology of logic, the declarative method amounts to expressing the desired properties of a problem class by sentences in a *logical theory*. The data of a particular problem instance corresponds naturally to a *partial interpretation (or structure)*. The solving process is to apply *model generation*, or more specifically *model expansion* (Mitchell & Ternovska, 2005), the task of finding a structure that expands the input partial structure and satisfies the theory. The resulting structure is a solution to the problem. Model generation/expansion, studied for example in the field of Knowledge Representation (KR) (Baral, 2003), is thus analogous to the task of solving constraint satisfaction problems, studied in CP, and that of generating answer sets of logic programs, studied in ASP.

The similarities between these areas go deeper and extend to the level of the used techniques. State-of-the-art approaches often follow a two-phase solving methodology. In a first phase, the input theory, in the rich language at hand, is reduced into a fragment of the language that is supported by some search algorithm. In the second phase, the search algorithm is applied to the reduced theory to effectively search for models. For example, model generation for the language MiniZinc (Nethercote et al., 2007) is performed by reducing to the ground language FlatZinc, for which search algorithms are available. Similarly, the language FO($\cdot$) (Denecker & Ternovska, 2008) is reduced to its propositional fragment PC($\cdot$) (see, e.g., Wittocx, Mariën, & Denecker, 2010), and ASP is reduced to propositional ASP (see, e.g., Gebser, Schaub, & Thiele, 2007). As the reduced theory is often in a ground fragment of the language, we refer to the resulting reduced theory as the *grounding* and to the first phase as the *grounding* phase (where quantifiers are instantiated with elements in the domain). In other fields, grounding is also referred to as flattening, unrolling, splitting or propositionalization. The solving methodology itself is generally referred to as *ground-and-solve*.

Grounding becomes a bottleneck as users turn to applications with large domains and complex constraints. Indeed, it is easy to see that the grounding size of an FO formula is exponential in the nesting depth of quantifiers and in the arity of predicates and polynomial in the size of the universe of discourse. There is an increasing number of applications where the size of the grounded theory is so large that it does not fit in memory. For example, Son, Pontelli, and Le (2014) discuss several ASP applications where the ground-and-solve approach turns out to be inadequate.

In this paper, we present a novel approach to remedy this bottleneck, called *lazy model expansion*, where the grounding is generated lazily (on-the-fly) during search, instead of up-front. The approach works by associating *justifications* to the non-ground parts of the theory. A valid justification for a non-ground formula is a recipe to expand a partial structure into a more precise (partial) structure that satisfies the formula. Of course, it is crucial that the recipe is a lot more compact than the grounding of the formula. Given a partial structure and a valid justification for each of the non-ground formulas, a (total) structure is obtained by extending the partial structure with the literals in the justifications of the non-ground formulas. Justifications are selected in such a way that this total structure is a model for the whole initial theory. Consequently, model generation can be limited to the grounded part of the theory; if a model is found for that part, it can be extended to a model of the whole theory. However, a new assignment during model generation can conflict with one of the justifications. In that case, an alternative justification needs to be sought. If none is found, the associated formula can be split in two parts, one part that is grounded and one part for which a valid justification is still available.

Example 1.1. Consider the `Sokoban` problem, a planning problem where a robot has to push blocks around on a 2-D grid to arrange them in a given goal configuration. A constraint on the move action is that the target position $p \in P$ of the moved block $b \in B$ is currently (at time $t \in T$) empty, which can be expressed as

$$\forall (t, b, p) \in T \times B \times P : move(b, p, t) \Rightarrow empty(p, t). \tag{1}$$

As it is not known in advance how many time steps are needed, one ideally wants to assume a very large or even infinite number of steps. Using ground-and-solve, this blows up the size of the grounding. Incremental grounding, iteratively extending the time domain until it is large enough to allow for a plan, has been developed to avoid the blow-up in the context of planning (Gebser et al., 2008). Our approach is more general and does not depend on the presence of one domain that can be incrementally increased.

Returning to the example, instead of grounding sentence (1), we associate with it a justification, a recipe to satisfy it. "Make $move(b, p, t)$ false for all b, p and t" is such a recipe. When the search finds a model for the grounded part of the problem that is not in conflict with the recipe, the model can be extended with the literals in the recipe to obtain a model of the whole theory. However, if the search would decide to move block b_1 to position p_1 at time t_1, a conflict is created with the recipe. To resolve it, the instance of sentence (1) that is in conflict with the partial model of the search is split off and sentence (1) is replaced by the equivalent sentences:

$$move(b_1, p_1, t_1) \Rightarrow empty(p_1, t_1) \tag{2}$$

$$\forall (t, b, p) \in T \times B \times P \backslash (t_1, b_1, p_1) : move(b, p, t) \Rightarrow empty(p, t) \tag{3}$$

Sentence (2) is grounded and passed to the search component which will use it to check that $empty(p_1, t_1)$ holds. Sentence (3) is non-ground and can be satisfied by the recipe "$move(b, p, t)$ is false except for $move(b_1, p_1, t_1)$". When the search makes more moves, more instances will be grounded, until the search finds a partial plan for the problem at hand. Then the literals in the recipe of the remaining non-ground formula —making $move(b, p, t)$ false for all instances of sentence (1) that have not been grounded— will complete the plan.

The main contributions of this paper are:

- A theoretical framework for *lazy model expansion*. By aiming at minimally instantiating quantified variables, it paves the way for a solution to the long-standing problem of handling quantifiers in search problems, encountered, e.g., in the fields of ASP (Lefèvre & Nicolas, 2009) and SAT Modulo Theories (Ge & de Moura, 2009). The framework also generalizes existing approaches that are related to the grounding bottleneck such as incremental domain extension (Claessen & Sörensson, 2003) and lazy clause generation (Ohrimenko, Stuckey, & Codish, 2009).

- A complete algorithm for lazy model expansion for the logic FO(ID), the extension of first-order logic (FO) with inductive definitions (a language closely related to ASP as shown in Denecker et al., 2012). This includes efficient algorithms to derive consistent sets of justifications and to maintain them throughout changes in a partial structure (e.g., during search).
- An implementation extending the IDP knowledge-base system (De Cat et al., 2014) and experiments that illustrate the power and generality of lazy grounding.

Lazy grounding is a new step in our ability to solve complex combinatorial problems. By avoiding the up-front grounding step of previous approaches, lazy grounding can ground enough of the problem to solve it. While our method is developed for the logic FO(ID), as will become clear, justifications are associated with rules, and these rules are very similar to the rules used by ASP systems. Hence, as discussed towards the end of the paper, our framework and algorithms can be applied also in the context of ASP.

The paper is organized as follows. In Section 2, the necessary background and notations are introduced. Formal definitions of lazy grounding with FO(ID) are presented in Section 3, followed by a presentation of the relevant algorithms and heuristics in Sections 4 and 5. Experimental evaluation is provided in Section 6, followed by a discussion on related and future work and a conclusion. A preliminary version of the paper appeared as the work of De Cat, Denecker, and Stuckey (2012) and of De Cat (2014, ch. 7).

2. Preliminaries

In this section, we provide the necessary background on the logic FO(ID), on the inference tasks model generation and model expansion for FO(ID) and on the ground-and-solve approach to model expansion.

2.1 FO(ID)

First, we define syntax and semantics of the logic FO(ID) (Denecker & Ternovska, 2008), the extension of first-order logic (FO) with inductive definitions. We assume familiarity with FO. Without loss of generality, we limit FO(ID) to the function-free fragment. Function symbols can always be eliminated using graph predicates (Enderton, 2001).

A (function-free) vocabulary Σ consists of a set of predicate symbols. Propositional symbols are 0-ary predicate symbols; these include the symbols $\top$ and $\bot$, denoting *true* and *false* respectively. Predicate symbols are usually denoted by P, Q, R; atoms by a, literals (atoms or their negation) by l; variables by x, y; and domain elements by d. With $\bar{e}$ we denote an ordered set of objects $e_1, \ldots, e_n$; with P/n a predicate P of arity n.

The methods for model generating developed below require that a (possibly infinite) domain D is given and fixed. Given a (function-free) vocabulary Σ, a *domain atom* is an atom of the form $P(\overline{d})$ with $P/n \in \Sigma$ and $\overline{d} \in D^n$, an n-tuple of domain elements. Likewise, we consider *domain literals*.

A structure $\mathcal{I}$ interpreting Σ consists of the domain D and an n-ary relation $P^{\mathcal{I}} \subseteq D^n$ for all predicate symbols $P/n \in \Sigma$. Alternatively, an n-ary relation can be viewed as a function $D^n \rightarrow \{\mathbf{t}, \mathbf{f}\}$. The propositional symbols $\top$ and $\bot$ are respectively interpreted as $\mathbf{t}$ and $\mathbf{f}$.

Model generation algorithms maintain *partial* structures and may (temporarily) find themselves in an *inconsistent* state, for example when a conflict arises. To represent such states, three-valued and four-valued structures $\mathcal{I}$ are introduced; they consist of the domain D and, for each n-ary predicate P in Σ, of a three- or four-valued relation $P^{\mathcal{I}}$. This is a function $D^n \to \{\mathbf{t}, \mathbf{f}, \mathbf{u}, \mathbf{i}\}$. A structure is *two-valued* if the range of its relations is $\{\mathbf{t}, \mathbf{f}\}$, *partial* or *three-valued* if the range is $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$) and four-valued in general. Thus, two-valued structures are also three-valued and four-valued. When unqualified, the term *structure* stands for the most general, four-valued case.

Given a fixed D and Σ, an alternative way to represent $\mathcal{I}$ is as a set S of domain literals. Indeed, there is a one-to-one correspondence between such sets S and Σ-structures $\mathcal{I}$ with domain D such that for a domain atom a, $a^{\mathcal{I}} = \mathbf{i}$ (inconsistent) if both a and $\neg a$ are in S, $a^{\mathcal{I}} = \mathbf{t}$ if only a is in S, $a^{\mathcal{I}} = \mathbf{f}$ if only $\neg a$ is in S and $a^{\mathcal{I}} = \mathbf{u}$ (unknown) otherwise. Hence, we may treat four-valued structures as sets of domain literals and vice versa. A structure is *inconsistent* if at least one domain atom is inconsistent.

A structure $\mathcal{I}$ of a vocabulary Σ can be naturally viewed as a structure of a larger vocabulary $\Sigma' \supset \Sigma$, namely by setting $a^{\mathcal{I}} = \mathbf{u}$ for any domain atom of a predicate in $\Sigma' \setminus \Sigma$.

For a set σ of predicate symbols, we use $\mathcal{I}|_\sigma$ to denote the restriction of $\mathcal{I}$ to the symbols of σ. For a set S of domain atoms, we use $\mathcal{I}|_S$ to denote the restriction of $\mathcal{I}$ to S: $a^{\mathcal{I}|_S} = a^{\mathcal{I}}$ if $a \in S$ and $a^{\mathcal{I}|_S} = \mathbf{u}$ otherwise. We call $\mathcal{I}$ a two-valued structure of S if $\mathcal{I}$ is two-valued on domain atoms of S and unknown otherwise.

The inverse v^{-1} of a truth value v is defined as follows: $\mathbf{t}^{-1} = \mathbf{f}$, $\mathbf{f}^{-1} = \mathbf{t}$, $\mathbf{u}^{-1} = \mathbf{u}$ and $\mathbf{i}^{-1} = \mathbf{i}$. The *truth* order $>_t$ on truth values is defined by $\mathbf{t} >_t \mathbf{u} >_t \mathbf{f}$ and $\mathbf{t} >_t \mathbf{i} >_t \mathbf{f}$. The *precision* order $>_p$ is defined by $\mathbf{i} >_p \mathbf{t} >_p \mathbf{u}$ and $\mathbf{i} >_p \mathbf{f} >_p \mathbf{u}$. Both orders are pointwise extended to arbitrary Σ-structures. We say that $\mathcal{I}$' is an *expansion* of $\mathcal{I}$ if $\mathcal{I}' \geq_p \mathcal{I}$, that is if for each domain atom a, $a^{\mathcal{I}'} \geq_p a^{\mathcal{I}}$. Viewing structures as sets of domain literals, this corresponds to $\mathcal{I}' \supseteq \mathcal{I}$.

We assume familiarity with the syntax of (function-free) FO. To facilitate the reasoning with partially grounded formulas, we deviate from standard FO and quantify over explicitly specified subsets of the domain D. This is denoted as $\exists x \in D' : \varphi$ and $\forall x \in D' : \varphi$, with $D' \subseteq D$. We sometimes abbreviate $\exists x_1 \in D_1 : \ldots \exists x_n \in D_n : \varphi$ as $\exists \overline{x} \in \overline{D} : \varphi$, and similarly for $\forall$. Given a formula φ, $\varphi[\overline{x}]$ indicates that $\overline{x}$ are the free variables of φ. Substitution of a variable x in formula φ by a term t is denoted by $\varphi[x/t]$. A *ground formula* (in domain D) is a formula without variables (hence without quantifiers). Similar properties and notations are used for *rules* (introduced below).

We denote by $voc(T)$ the set of all predicate symbols that occur in theory T. For a structure $\mathcal{I}$, $voc(\mathcal{I})$ is the set of symbols interpreted by $\mathcal{I}$. Unless specified otherwise, theories and structures range over the vocabulary Σ.

The language FO(ID) extends FO with (inductive) *definitions*. A theory in FO(ID) is a (finite) set of sentences and definitions. A definition Δ is a (finite) set of rules of the form $\forall \overline{x} \in \overline{D} : P(x_1, \ldots, x_n) \leftarrow \varphi$, with P a predicate symbol and φ an FO formula. The atom $P(\overline{x})$ is referred to as the *head* of the rule and φ as the *body*. Given a rule r, we let $head(r)$ and $body(r)$ denote respectively the head and the body of r. Given a definition Δ, a domain atom $P(\overline{d})$ is *defined* by Δ if there exists a rule $\forall \overline{x} \in \overline{D} : P(\overline{x}) \leftarrow \varphi$ in Δ such that $\overline{d} \in \overline{D}$. Otherwise $P(\overline{d})$ is *open* in Δ. A domain literal $\neg P(\overline{d})$ is defined by Δ if $P(\overline{d})$

is defined by Δ. The sets of defined and open domain atoms of Δ are denoted as $\mathit{defined}(\Delta)$ and $\mathit{open}(\Delta)$, respectively.

Without loss of generality, we assume that in any definition a domain atom is defined by at most one rule. Technically, this means that rules $\forall \overline{x} \in \overline{D}_1 : P(\overline{x}) \leftarrow \varphi_1$, $\forall \overline{x} \in \overline{D}_2 : P(\overline{x}) \leftarrow \varphi_2$ are pairwise disjunct, that is $\overline{D}_1 \cap \overline{D}_2 = \emptyset$. Rules can always be made disjunct by transforming them in $\forall \overline{x} \in \overline{D}_1 \cap \overline{D}_2 : P(\overline{x}) \leftarrow \varphi_1 \vee \varphi_2$, $\forall \overline{x} \in \overline{D}_1 \setminus \overline{D}_2 : P(\overline{x}) \leftarrow \varphi_1$, $\forall \overline{x} \in \overline{D}_2 \setminus \overline{D}_1 : P(\overline{x}) \leftarrow \varphi_2$.

2.1.1 Model Semantics

The semantics of FO(ID) is a two-valued model semantics. Nevertheless, we introduce concepts of three- and four-valued semantics which are useful in defining the semantics of definitions and in formalizing lazy grounding. We use the standard four-valued truth assignment function, defined by structural induction for pairs of FO domain formulas φ and structures $\mathcal{I}$ that interpret φ:

- $P(\overline{d})^{\mathcal{I}} = P^{\mathcal{I}}(\overline{d}^{\mathcal{I}})$,
- $(\psi \wedge \phi)^{\mathcal{I}} = min_{<_t}(\psi^{\mathcal{I}}, \phi^{\mathcal{I}})$,
- $(\psi \vee \phi)^{\mathcal{I}} = max_{<_t}(\psi^{\mathcal{I}}, \phi^{\mathcal{I}})$,
- $(\neg\psi)^{\mathcal{I}} = (\psi^{\mathcal{I}})^{-1}$,
- $(\exists x \in D : \psi)^{\mathcal{I}} = max_{<_t}(\{\psi[x/d]^{\mathcal{I}} \mid d \in D\})$,
- $(\forall x \in D : \psi)^{\mathcal{I}} = min_{<_t}(\{\psi[x/d]^{\mathcal{I}} \mid d \in D\})$.

The assignment function is monotonic in the precision order: if $\mathcal{I} \leq_p \mathcal{I}'$, then $\varphi^{\mathcal{I}} \leq_p \varphi^{\mathcal{I}'}$. Hence, if a formula is true in a partial structure, it is true in all two-valued expansions of it. Also, if $\mathcal{I}$ is two-valued (respectively three-valued, four-valued) then $\varphi^{\mathcal{I}}$ is two-valued (respectively three-valued, four-valued).

A structure $\mathcal{I}$ is a *model* of / *satisfies* a sentence φ (notation $\mathcal{I} \models \varphi$) if $\mathcal{I}$ is two-valued and $\varphi^{\mathcal{I}} = \mathbf{t}$. The satisfaction relation can be defined for definitions as well. The semantics of definitions is based on the parametrized well-founded semantics, an extension of the well-founded semantics of logic programs informally described first in the work of Van Gelder (1993), and formally defined for FO(ID)'s definitions by Denecker (2000). This semantics formalizes the informal semantics of rule sets as (inductive) definitions (Denecker, 1998; Denecker, Bruynooghe, & Marek, 2001; Denecker & Vennekens, 2014). A structure $\mathcal{I}$ is a *model* of / *satisfies* a definition Δ (notation $\mathcal{I} \models \Delta$) if $\mathcal{I}$ is two-valued and is the well-founded model —denoted as $wf_\Delta(\mathcal{I}|_{open(\Delta)})$— of Δ in the structure $\mathcal{I}|_{open(\Delta)}$ (Denecker & Ternovska, 2008). In case $wf_\Delta(\mathcal{I}|_{open(\Delta)})$ is not two-valued, Δ has no model expanding $\mathcal{I}|_{open(\Delta)}$. A structure $\mathcal{I}$ satisfies a theory $\mathcal{T}$ if $\mathcal{I}$ is two-valued and $\mathcal{I}$ is a model of all sentences and definitions in $\mathcal{T}$. In the next subsection, we present a formalization of the well-founded semantics using the notion of *justification*.

According to FO(ID)'s methodology, (formal) definitions are used to express informal definitions. In the work of Denecker and Vennekens (2014), it was shown that FO(ID) definitions offer a uniform representation of the most important types of informal definitions and that expressing informal definitions leads to rule sets that are *total*. Formally, a definition Δ is called *total* if the well-founded model of Δ in each two-valued structure $\mathcal{I}$ of $open(\Delta)$ is two-valued (Denecker & Ternovska, 2008). In general, totality is undecidable; however broad, syntactically defined classes of definitions have been proven to be total (e.g., non-recursive, positive, stratified and locally stratified definitions, see Denecker & Ternovska, 2008). Inspection of current FO(ID) applications shows that in practice, non-total definitions occur rarely and almost always contain a modeling error. Also, in most cases totality can be established through a simple syntactic check. Totality can be usefully exploited during computation. The lazy grounding techniques introduced below exploit totality and should be applied only to total definitions. This restriction matches with FO(ID)'s design and methodology and, in practice, this does not impose a strong limitation. In case the input theory does contain definitions that are not known to be total, all is not lost: those definitions can be grounded completely up-front, in which case lazy grounding can be applied safely to the remaining sentences and total definitions in the input.

Equivalence. Two theories T and T', which can be over different vocabularies, are Σ-*equivalent* if each model of T restricted to Σ can be expanded to a model of T' and vice versa. Two theories T and T' are *strongly* Σ-*equivalent* if the above expansions are also unique. By extension, (strong) Σ-equivalence *in a structure* $\mathcal{I}$ is defined similarly: if each model of T expanding $\mathcal{I}$ can be expanded to a model of T' expanding $\mathcal{I}$ and vice versa; to obtain strong equivalence, these expansions have to be unique. From a theory T, we often derive a *strongly* $voc(T)$-equivalent theory T' in a given structure $\mathcal{I}$. Such transformations preserve satisfiability and number of models *and* each model of T' can be directly mapped to a model of T by projection on $voc(T)$.

Canonical theories. To simplify the presentation, the lazy grounding techniques are presented here for theories of the form $\{P_{\mathcal{T}}, \Delta\}$, with $P_{\mathcal{T}}$ a propositional symbol, and Δ a single definition with function-free rules. This is without loss of generality. First, as mentioned above, standard techniques (Enderton, 2001) allow one to make a theory function-free. Second, multiple definitions can always be combined into one as described by Denecker and Ternovska (2008) and Mariën, Gilis, and Denecker (2004). This is achieved by renaming defined predicates in some of the definitions, merging all rules into one set and adding equivalence constraints between predicates and their renamings. Third, the theory $\mathcal{T} = \{\varphi_1, \ldots, \varphi_n, \Delta\}$ resulting from the previous step can be translated to the strongly $voc(\mathcal{T})$-equivalent theory $\{P_{\mathcal{T}}, \Delta \cup \{P_{\mathcal{T}} \leftarrow \varphi_1 \wedge \cdots \wedge \varphi_n\}\}$ with $P_{\mathcal{T}}$ a new propositional symbol. This transformation results in a ground set of sentences and a definition consisting of a set of (ground and non-ground) rules, so lazy grounding has only to cope with non-ground rules. Furthermore, we assume that rule bodies are in negation normal form (negation only occurs in front of atoms) and that, for each defined domain atom $P(\overline{d})$, there is a unique rule $\forall \overline{x} \in \overline{D} : P(\overline{x}) \leftarrow \varphi \in \Delta$ such that $\overline{d} \in \overline{D}$.

The methods proposed below can be extended to full FO(ID) with functions, and such extended methods have been implemented in our system. However, this introduces a number of rather irrelevant technicalities which we want to avoid here.

2.1.2 Justifications

We assume the presence of a domain D and a canonical theory $\mathcal{T} = \{\varphi, \Delta\}$ as explained above. Recall, structures with domain D correspond one-to-one to sets of domain literals.

Definition 2.1 (Direct justification)**.** A *direct justification* for a defined domain literal $P(\overline{d})$ (respectively $\neg P(\overline{d})$) is a consistent non-empty set S of domain literals such that, for the rule $\forall \overline{x} \in \overline{D} : P(\overline{x}) \leftarrow \varphi$ of Δ such that $\overline{d} \in \overline{D}$, it holds that $\varphi[\overline{x}/\overline{d}]^S = \mathbf{t}$ (respectively $\varphi[\overline{x}/\overline{d}]^S = \mathbf{f}$).

Any consistent superset S' of a direct justification S of $P(\overline{d})$ is a direct justification as well. Indeed, a body $\varphi[\overline{x}/\overline{d}]$ true in S is true in the more precise S'. Also, a direct justification S is not empty by definition; if φ is true in every structure, then a minimal direct justification is $\{\top\}$.

Example 2.2. Consider a domain $D = \{d_1, \ldots, d_n\}$ and the definition Δ

$$\left\{ \begin{array}{ll} \forall x \in D : P(x) & \leftarrow Q(x) \vee R(x) \\ \forall x \in D : Q(x) & \leftarrow P(x) \end{array} \right\}$$

A direct justification for $Q(d_i)$ is $\{P(d_i)\}$ and for $\neg Q(d_i)$ is $\{\neg P(d_i)\}$. Both domain literals have many other direct justifications, but those are the unique minimal ones under the subset relation. Minimal direct justifications for $P(d_i)$ are both $\{Q(d_i)\}$ and $\{R(d_i)\}$ while the only minimal direct justification for $\neg P(d_i)$ is $\{\neg Q(d_i), \neg R(d_i)\}$. Atoms $R(d_i)$ are open and have no direct justification.

A (directed) graph G is a pair $\langle V, E\rangle$ of a set V of nodes and a set E of directed edges, i.e., ordered pairs (v_i, v_j) of nodes. For any node $v \in V$, we denote by $G(v)$ the set of children of v, i.e., $G(v) = \{w \mid (v, w) \in E\}$.

Definition 2.3 (Justification)**.** A *justification* over a definition Δ is a graph J over the set of domain literals of Δ such that for each domain literal l, $J(l)$ is either empty or a direct justification of l.

Thus, a justification is a graph that encodes for every defined domain literal none or one direct justification. In the sequel we say that J is *defined in l* if $J(l) \neq \emptyset$. A justification is denoted as a set of pairs $l \rightarrow S$, with S a direct justification of l.

Definition 2.4 (Justification subgraph)**.** Let J be a justification over Δ. The justification *for a literal l* is the subgraph J_l of nodes and edges of J reachable from l. The justification *for a set of literals L* is the subgraph J_L of nodes and edges of J reachable from any $l \in L$.

A justification J over Δ is *total for l* if J is defined in each literal that is reachable from l and defined in Δ; it is *total for a set of literals L* if it is total for each literal in L. A justification J is *consistent with* a structure $\mathcal{I}$ if $\mathcal{I}$ is consistent and none of the literals for which J is defined is false in $\mathcal{I}$.

If J is total for l, then the leaves of J_l are open domain literals.

Definition 2.5. A path in a justification J is a sequence $l_0 \to l_1 \to \ldots$ such that, if $l_i \to l_{i+1}$, then there is an edge from l_i to l_{i+1} in J. A path is *positive* if it consists of only positive literals; it is *negative* if it consists of only negative literals; it is *mixed* otherwise. A *cycle* in a justification J is a set of domain literals on a path in J that starts and ends in the same domain literal. A cycle is positive (respectively, negative) if all domain literals are positive literals (respectively, negative literals); otherwise the cycle is mixed.

An infinite path may be cyclic or not. If D is finite, every infinite path is cyclic.

Intuitively, a justification J containing a domain literal l provides an argument for the truth of l. The strength of this argument depends on the truth of the leaves and on the infinite paths and cycles in J_l. If all leaves are true and every infinite path is negative, J_l provides the argument that l is true. If a leaf is false or unknown, or J_l contains a positive or mixed loop, the argument for l is weak. Notice that other justifications for l may still argue l's truth.

Definition 2.6 (Justifies)**.** We say that a defined literal l is *well-founded* in the justification J if every infinite path in J_l is negative. Otherwise l is *unfounded* in J.

A justification J over Δ *justifies* a set of literals L defined in Δ (the set L of literals *has a justification* J) if (**i**) J_L is total for L; (**ii**) each literal of L is well-founded in J; (**iii**) the set of literals in J_L is consistent.

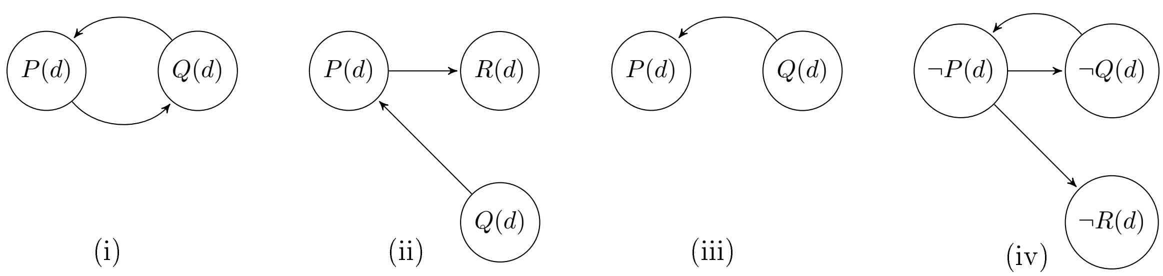

Figure 1: Justifications for definition Δ in Example 2.2, with $d \in D$.

Example 2.7. In Figure 1, we show a few possible justifications (ordered (i)-(iv) from left to right) over definition Δ in Example 2.2 that contain the defined domain atoms $P(d)$ and $Q(d)$ ($d \in D$). Justification (ii) justifies $P(d)$ and $Q(d)$ and (iv) justifies $\neg P(d)$ and $\neg Q(d)$; (iii), however, is not total for $P(d)$ nor $Q(d)$ and (i) has a positive cycle and is unfounded for both $P(d)$ and $Q(d)$.

The relationship between justifications and the well-founded semantics has been investigated in different publications (Denecker & De Schreye, 1993, 1992; Mariën, 2009). Below we recall the results on which this paper relies. The first result states that if J justifies all literals in L, then any model $\mathcal{I}$ of Δ in which the leaves of J_L are true, satisfies all literals in L and in J_L.

Proposition 2.8. *If J is a justification over Δ that justifies a set of domain literals L then all literals in J_L are true in every model of Δ in which the (open) leaves of J_L are true.*

For an interpretation $\mathcal{I}_{open}$ that is two-valued for $open(\Delta)$, the well-founded model $wf_\Delta(\mathcal{I}_{open})$ can be computed in time polynomial in the size of the domain, as shown by Chen and Warren (1996). In general, $wf_\Delta(\mathcal{I}_{open})$ is a three-valued structure. If $wf_\Delta(\mathcal{I}_{open})$ is two-valued, then it is the unique model of Δ that expands $\mathcal{I}_{open}$; otherwise, Δ has no model that expands $\mathcal{I}_{open}$. The above proposition follows from the fact that if a justification J justifies L and all leaves of J are true in $\mathcal{I}_{open}$, then all literals of L are true in $wf_\Delta(\mathcal{I}_{open})$.

Example 2.9 (Continued from Example 2.7)**.** Justification (ii) justifies $L = \{Q(d)\}$ and has a unique open leaf $R(d)$. For any structure $\mathcal{I}_{open}$ interpreting the open predicates of Δ, if $R(d)$ is true in $\mathcal{I}_{open}$, then $Q(d)$ is true in $wf_\Delta(\mathcal{I}_{open})$. In particular, in any model of Δ in which $R(d)$ is true, $Q(d)$ is true.

Proposition 2.10. *If $\mathcal{I}$ is a model of Δ, then a justification J over Δ exists that consists of literals true in $\mathcal{I}$, is defined for all defined domain literals true in $\mathcal{I}$ and justifies each of them.*

Corollary 2.11. *In case Δ is total, if a justification J over Δ justifies a set of domain literals L, then every two-valued $open(\Delta)$-structure consistent with J_L can be extended in a unique way to a model of Δ that satisfies all literals of L.*

Hence, for a canonical theory $\{P_\mathcal{T}, \Delta\}$ (recall, Δ is total), the theory is satisfiable iff a justification J exists that justifies $P_\mathcal{T}$.

2.2 Generating Models

Model generation is the inference task that takes as input a theory $\mathcal{T}$ and returns as output a model of $\mathcal{T}$. Model Expansion (MX) was defined by Mitchell et al. (2006) as the inference task that takes as input a theory $\mathcal{T}$ over vocabulary Σ and a two-valued structure $\mathcal{I}$ over a subvocabulary of Σ, and returns an expansion $\mathcal{M}$ of $\mathcal{I}$ that satisfies $\mathcal{T}$. Here, it will be the more general inference problem as defined by Wittocx, Mariën, and Denecker (2008) that takes as input a (potentially partial) structure $\mathcal{I}$ over Σ, and returns an expansion $\mathcal{M}$ of $\mathcal{I}$ that satisfies $\mathcal{T}$.

As already mentioned, the state-of-the-art approach to model expansion in FO(ID) is (similar to ASP) grounding $\mathcal{T}$ in the context of $\mathcal{I}$ and afterwards applying search to the resulting ground theory. The latter can, e.g., be accomplished by the SAT(ID) search algorithm (Mariën et al., 2008).

Below, we present the grounding algorithm that is the basis of the lazy MX algorithm. We assume familiarity with the basic Conflict-Driven Clause-Learning (CDCL) algorithm of SAT solvers (Marques Silva, Lynce, & Malik, 2009).

2.2.1 Grounding

For an overview of intelligent grounding techniques in FO(ID), we refer the reader to the work of Wittocx, Denecker, and Bruynooghe (2013) and of Wittocx et al. (2010). Below we present the basic principle.

A grounder takes as input a theory $\mathcal{T}$ over vocabulary Σ, a partial structure $\mathcal{I}$ with domain D, interpreting at least $\top$ and $\bot$, and returns a ground theory $\mathcal{T}'$ that is strongly

Σ-equivalent with $\mathcal{T}$ in $\mathcal{I}$. Theory $\mathcal{T}'$ is then called a *grounding* of $\mathcal{T}$ given $\mathcal{I}$. Recall that we assume that $\mathcal{T}$ is a canonical theory of the form $\{P_{\mathcal{T}}, \Delta\}$.

One way to compute the grounding is using a top-down process on the theory, iteratively applying grounding steps to direct subformulas of the rule or formula at hand. The grounding algorithm may replace subformulas by new predicate symbols as follows. Let $\varphi[\overline{x}]$ be a formula in $\mathcal{T}$ and let $\overline{D}$ be the domains of $\overline{x}$. A *Tseitin transformation* replaces φ by the atom $T_\varphi(\overline{x})$, with T_φ a new $|\overline{x}|$-ary predicate symbol called a *Tseitin* symbol,[1] and extends Δ with the rule $\forall \overline{x} \in \overline{D} : T_\varphi(\overline{x}) \leftarrow \varphi$. The new theory is strongly Σ-equivalent to the original one (Vennekens et al., 2007).

The procedure one_step_ground, outlined in Figure 1, performs one step in the grounding process. Called with a formula or rule φ in canonical form, the algorithm replaces all direct subformulas with Tseitin symbols and returns a pair consisting of a ground part G (rules or formulas) and a possibly non-ground part R (rules). If φ is a formula, then G consists of ground formulas. Replacing φ by the returned ground formulas and extending Δ with the returned rules produces a theory that is strongly $voc(\mathcal{T})$-equivalent to the original. If φ is a rule from Δ, G consists of ground rules, and replacing φ by both sets of returned rules results again in a theory that is strongly $voc(\mathcal{T})$-equivalent to the original.

Algorithm 1: The one_step_ground algorithm.

Function one_step_ground (*formula or rule* φ)
  switch φ **do**
    case $[\neg]P(\overline{d})$ **return** $\langle\{\varphi\}, \emptyset\rangle$;
    case $P(\overline{d}) \leftarrow \psi$
      $\langle G, \Delta\rangle :=$ one_step_ground(ψ);
      return $\langle\{P(\overline{d}) \leftarrow \bigwedge_{g\in G} g\}, \Delta\rangle$;
    case $\psi_1 \vee \ldots \vee \psi_n$
      return $\langle\{\bigvee_{i\in[1,n]} T_{\psi_i}\}, \{T_{\psi_i} \leftarrow \psi_i \mid i \in [1,n]\}\rangle$;
    case $\psi_1 \wedge \ldots \wedge \psi_n$
      return $\langle\{T_{\psi_i} \mid i \in [1,n]\}, \{T_{\psi_i} \leftarrow \psi_i \mid i \in [1,n]\}\rangle$;
    case $\forall \overline{x} \in \overline{D} : P(\overline{x}) \leftarrow \psi$
      return $\langle\emptyset, \{P(\overline{x})[\overline{x}/\overline{d}] \leftarrow \psi[\overline{x}/\overline{d}] \mid \overline{d} \in \overline{D}\}\rangle$;
    case $\exists \overline{x} \in \overline{D} : \psi[\overline{x}]$
      return $\langle\{\bigvee_{d\in D} T_{\psi[\overline{x}/\overline{d}]}\}, \{T_{\psi[\overline{x}/\overline{d}]} \leftarrow \psi[\overline{x}/\overline{d}] \mid \overline{d} \in \overline{D}\}\rangle$;
    case $\forall \overline{x} \in \overline{D} : \psi[\overline{x}]$
      return $\langle\{T_{\psi[\overline{x}/\overline{d}]} \mid \overline{d} \in \overline{D}\}, \{T_{\psi[\overline{x}/\overline{d}]} \leftarrow \psi[\overline{x}/\overline{d}] \mid \overline{d} \in \overline{D}\}\rangle$;

Grounding a theory then boils down to applying one_step_ground on the sentence $P_{\mathcal{T}}$ (which copies $P_{\mathcal{T}}$ to the ground part) and on each rule of the theory and repeatedly applying one_step_ground on the returned rules R (all returned sentences and rules in G are ground). We use ground to refer to the algorithm for this overall process.

1. Tseitin (1968) introduced such symbols as part of his normal form transformation.

Various improvements exist, such as returning $\top/\bot$ for atoms interpreted in $\mathcal{I}$ and returning $\bot$ from conjunctions whenever a false conjunct is encountered (analogously for disjunctions and quantifications).

Also, algorithm `one_step_ground` introduces a large number of Tseitin symbols. State-of-the-art grounding algorithms use a number of optimizations to reduce the number of such symbols. As these optimizations are not directly applicable to the techniques presented in this paper, we start from the naive `one_step_ground` algorithm. In Section 5, we present an optimized version of `one_step_ground` that introduces fewer Tseitin symbols and hence results in smaller groundings.

3. Lazy Grounding and Lazy Model Expansion

We use the term *lazy grounding* to refer to the process of partially grounding a theory and the term *lazy model expansion* (lazy MX) for the process that interleaves lazy grounding with model expansion over the grounded part. In Section 3.1, we formalize a framework for lazy model expansion of FO(ID) theories; in Section 3.2, we formalize the instance of this framework that is the basis of our current implementation; in Section 3.3, we illustrate its operation.

3.1 Lazy Model Expansion for FO(ID) Theories

Given a canonical theory $\mathcal{T} = \{P_\mathcal{T}, \Delta\}$ and an input structure $\mathcal{I}_{in}$, models expanding $\mathcal{I}_{in}$ are searched for by interleaving lazy grounding with search on the already grounded part. We first focus on the lazy grounding.

Apart from the initial step that moves $P_\mathcal{T}$ to the grounded part, the input of each step consists of a set of rules still to be grounded, an already grounded theory and a three-valued structure that is an expansion of the initial input structure.

Each subsequent grounding step can replace non-ground rules by ground rules and might introduce new rules. Hence, the state of the grounding includes a set Δ_g of ground rules and a set Δ_d (the *delayed definition*) of (possibly) non-ground rules. The definitions have the property that $\Delta_\mathrm{g} \cup \Delta_\mathrm{d}$ (in what follows abbreviated as Δ_gd) is $voc(\Delta)$-equivalent with the original definition Δ and hence, Δ_gd is total. The grounding procedure will guarantee that, at all times, Δ_g and Δ_d are total.

Given a partial structure $\mathcal{I}_{in}$ and the rule sets Δ_g and Δ_d, the key idea behind lazy model expansion is (**i**) to use a search algorithm to search for a model $\mathcal{I}$ of Δ_g that is an expansion of $\mathcal{I}_{in}$ in which $P_\mathcal{T}$ is true; (**ii**) to maintain a justification J such that the literals true in $\mathcal{I}$ and defined in Δ_d are justified over Δ_gd and that J is consistent with $\mathcal{I}$; (**iii**) to interleave steps (**i**) and (**ii**) and to move parts of Δ_d to Δ_g when some literal defined in Δ_d that needs to be justified cannot be justified.

Thus, to control lazy model expansion, it suffices to maintain a state $\langle \Delta_\mathrm{g}, \Delta_\mathrm{d}, J, \mathcal{I} \rangle$ consisting of the grounded rules Δ_g, the rules Δ_d yet to be grounded, a justification J, and a three-valued structure $\mathcal{I}$. Initially, $\mathcal{I}$ is $\mathcal{I}_{in}$, Δ_d is Δ, $\Delta_\mathrm{g} = \emptyset$, and J is the empty graph.

Lazy model expansion searches over the space of *acceptable* states.

Definition 3.1 (Acceptable state)**.** A tuple $\langle \Delta_\mathrm{g}, \Delta_\mathrm{d}, J, \mathcal{I} \rangle$ of a theory with an atomic sentence $P_\mathcal{T}$, a total definition Δ, and an input structure $\mathcal{I}_{in}$ is an *acceptable* state if (**i**) Δ_gd, Δ_g

and Δ_{d} are total definitions and Δ_{gd} is strongly $voc(\Delta)$-equivalent with Δ, (**ii**) no domain atom is defined in both Δ_{g} and Δ_{d}, (**iii**) J is a justification over Δ_{gd}, (**iv**) $\mathcal{I}$ is an expansion of $\mathcal{I}_{in}$, (**v**) the set L of literals true in $\mathcal{I}$ and defined in Δ_{d} is justified by J, and (**vi**) J_L, the justification of the literals in L, is consistent with $\mathcal{I}$.

Example 3.2. Consider the theory $\{P_{\mathcal{T}}, \Delta\}$, with Δ the definition

$$\left\{\begin{array}{l} P_{\mathcal{T}} \leftarrow T_1 \lor T_2 \lor T_3. \\ T_1 \leftarrow \forall x \in D : Q(x). \\ T_2 \leftarrow \forall x \in D : R(x). \\ T_3 \leftarrow \exists x \in D : \neg Q(x). \end{array}\right\}$$

Let $\mathcal{I}$ be the structure $\{P_{\mathcal{T}}, T_1\}$ (hence, T_2 and T_3 are unknown), and Δ_{g} and Δ_{d} the definitions consisting of the first rule and the remaining rules, respectively. Furthermore, let J be $\{T_1 \to \{Q(d) \mid d \in D\}\}$. The tuple $\langle \Delta_{\mathrm{g}}, \Delta_{\mathrm{d}}, J, \mathcal{I}\rangle$ is then an acceptable state. Indeed, T_1 is the only literal in $\mathcal{I}$ that is defined in Δ_{d} and it is justified by J.

As already said, the lazy model expansion algorithm starts from the initial state $\Delta_g = \emptyset, \Delta_d = \Delta, J = \emptyset, \mathcal{I} = \mathcal{I}_{in}$, which is acceptable if defined literals are unknown in $\mathcal{I}_{in}$. In each state, it either refines $\mathcal{I}$ by propagation or choice, or it backjumps. If the resulting state is unacceptable, a repair operation restores acceptability; these steps are described in Section 3.2. The algorithm tries to compute an acceptable state in which $P_{\mathcal{T}}$ is justified in Δ_{gd}. By Corollary 2.11, this would entail that a model of $\mathcal{T}$ exists; it can be computed efficiently through well-founded model computation. In intermediate states, the justification may be non-total for $P_{\mathcal{T}}$, contain unfounded literals, or be inconsistent.

Note that, in (**iii**), the justification must be over Δ_{gd}. Indeed, assume some literal l is justified over Δ_{d}. Its justification graph can have a leaf that is defined in Δ_{g} and that depends positively or negatively on l. Then every attempt to extend this justification graph to a total justification graph that justifies l over Δ_{gd} might fail, e.g., because of a forbidden cycle. Consider, e.g., the definitions $\Delta_{\mathrm{g}} = \{P \leftarrow Q\}$ and $\Delta_{\mathrm{d}} = \{Q \leftarrow P\}$. In that case, it would not be correct to take P as justification for Q being true, even though it is a valid justification within Δ_{d}. Indeed, no model exists that justifies Q in the full definition Δ_{gd}.

Proposition 3.3. *Let $\langle \Delta_g, \Delta_d, J, \mathcal{I}\rangle$ be an acceptable state. Δ_{gd} has a well-founded model that expands the literals that are true in $\mathcal{I}$ and defined in the (delayed) definition Δ_d.*

Proof. Let L be the set of literals true in $\mathcal{I}$ and defined in Δ_{d}. As the state is acceptable, J justifies the literals of L. Hence, by Corollary 2.11, there exists a well-founded model that expands L. □

Example 3.4 (Continued from Example 3.2)**.** The well-founded evaluation, after assigning $\top$ to the open literals of J (i.e., to $\{Q(d) \mid d \in D\}$), derives that T_1 is true. Moreover, because $\mathcal{I}$ is a model of Δ_{g}, $P_{\mathcal{T}}$ is also true in such a well-founded model. Note that R can be interpreted randomly, as no R-atoms occur in $\mathcal{I}$ or J.

The following theorem states when the obtained expansion is also a model of $\mathcal{T}$.

Theorem 3.5. *Let $\langle \Delta_g, \Delta_d, J, \mathcal{I} \rangle$ be an acceptable state of a theory $\mathcal{T} = (P_{\mathcal{T}}, \Delta)$ with input structure $\mathcal{I}_{in}$ such that $P_{\mathcal{T}}$ is true in $\mathcal{I}$ and $\mathcal{I}|_{voc(\Delta_g)}$ is a model of Δ_g. Then there exists a model $\mathcal{M}$ of $\mathcal{T}$ that expands $\mathcal{I}|_{voc(\Delta_g)}$.*

Proof. $\mathcal{I}|_{voc(\Delta_{\mathrm{g}})}$ is a model of Δ_{g}. It follows from Proposition 2.10 that there exists a justification J_g over Δ_{g} that justifies every true defined literal of Δ_{g} and that consists of only domain literals true in $\mathcal{I}|_{voc(\Delta_{\mathrm{g}})}$. We now have two justifications: J and J_g. We combine them in one J_c as follows: for each defined literal l of Δ_{gd}, if J is defined in l, we set $J_c(l) = J(l)$; otherwise, we set $J_c(l) = J_g(l)$. As J_c takes edges from either J or J_g for each defined literal, it is a justification for Δ_{gd}.

We verify that J_c justifies $P_{\mathcal{T}}$. First, it is total in $P_{\mathcal{T}}$. Indeed, any path from $P_{\mathcal{T}}$ either consists of literals defined in Δ_{g}, and then it is a branch of the total J_g over Δ_{g}, or it passes to a literal l' defined in Δ_{d}, which is justified by J according to condition ($\mathbf{v}$) and hence $(J_c)_{l'} = J_{l'}$ is total. As such, from $P_{\mathcal{T}}$ we cannot reach a defined literal of Δ_{gd} in which J_c is undefined. Second, J_c does not contain unfounded literals starting from $P_{\mathcal{T}}$. This is because any path from $P_{\mathcal{T}}$ is either a path in J_g (so well-founded as it justifies Δ_{g}) or it has a tail in J (well-founded by property ($\mathbf{v}$)). Finally, the set of literals reachable from $P_{\mathcal{T}}$ in J_c is consistent. Also this we can see if we look at paths in J_c from $P_{\mathcal{T}}$: at first we follow J_g which consists of true literals in $\mathcal{I}$, then we may get into a path of J which contains literals that are consistent with $\mathcal{I}$. In any case, it is impossible to reach both a literal and its negation.

It follows from Proposition 2.8 that there exists a model of Δ_{gd} that expands $\mathcal{I}|_{voc(\Delta_{\mathrm{g}})}$ and in which $P_{\mathcal{T}}$ is true. Since Δ_{gd} is strongly equivalent with Δ, the proposition follows. $\square$

Recall that effectively computing such a model $\mathcal{M}$ can be achieved by well-founded evaluation of Δ_{gd}, with polynomial data complexity, starting from any two-valued *open*(Δ_{gd})-structure expanding $\mathcal{I}|_{voc(\Delta_{\mathrm{g}})}$ (Chen & Warren, 1996).

In the above theorem, it is required that $\mathcal{I}$ is a model of Δ_{g}. Actually, we do not need to compute a two-valued model of Δ_{g}. It suffices to search for a partial structure and a justification that justifies $P_{\mathcal{T}}$. So, we can relax this requirement at the expense of also maintaining justifications for literals true in $\mathcal{I}$ and defined in Δ_{g}.

Corollary 3.6. *Let $\langle \Delta_g, \Delta_d, J, \mathcal{I} \rangle$ be an acceptable state of a theory $\mathcal{T} = \{P_{\mathcal{T}}, \Delta\}$ with input structure $\mathcal{I}_{in}$ such that $P_{\mathcal{T}}$ is true in $\mathcal{I}$ and J justifies $P_{\mathcal{T}}$ over Δ_{gd}. Then there exists a model $\mathcal{M}$ of $\mathcal{T}$ that expands $\mathcal{I}|_S$ with S the set of defined literals in $J_{P_{\mathcal{T}}}$.*

Failure to find a model of Δ_{g} expanding $\mathcal{I}_{in}$ in which $P_{\mathcal{T}}$ is true implies the lack of models of $\mathcal{T}$ expanding $\mathcal{I}_{in}$. Indeed, if Δ_{g} has no model expanding $\mathcal{I}_{in}$, then it has an unsatisfiable core, i.e., a set of rules from Δ_{g} such that no model exists that expands $\mathcal{I}_{in}$. Hence, it is also an unsatisfiable core for $\mathcal{T} = (P_{\mathcal{T}}, \Delta)$. To find an unsatisfiable core, one can, for example, use techniques described by Torlak, Chang, and Jackson (2008).

3.2 Practical Justification Management for FO(ID) Theories

Roughly speaking, our lazy model expansion framework consists of two components. On the one hand, a standard model expansion algorithm that operates on $\{P_{\mathcal{T}}, \Delta_{\mathrm{g}}\}$ and, on the other hand, a justification manager that maintains a justification over Δ_{gd} and lazily grounds Δ_d. Lazy model expansion performs search over the space of acceptable states and aims at reaching a state where Theorem 3.5 (or Corollary 3.6) is applicable. To avoid slowing down the search during model expansion, the work done by the justification manager and the lazy grounding must be limited. To achieve this, we have designed a system in which the justification manager has no access to the grounded definition Δ_g and need not restore its state when the search algorithm backtracks over the current structure $\mathcal{I}$. The justification manager only has access to $\mathcal{I}$ and maintains justifications that are restricted to Δ_d. In particular, a literal defined in Δ_g is not allowed in a direct justification. Our justification manager maintains the following properties:

- Literals in direct justifications are either open in Δ_gd or defined in Δ_d.
- All direct justifications in J are kept consistent with each other and with the current structure $\mathcal{I}$.
- The justification graph defined by J has no unfounded literals and is total.

To distinguish acceptable states that meet these additional requirements from acceptable states as defined in Definition 3.1, we call them *default acceptable states*; we define them as:

Definition 3.7 (Default acceptable state)**.** A state $\langle \Delta_\text{g}, \Delta_\text{d}, J, \mathcal{I}\rangle$ is a default acceptable state if it is an acceptable state and, in addition, (**i**) literals in direct justifications are either open in Δ_gd or defined in Δ_d, and (**ii**) J justifies the set of all literals for which J is defined.

It follows that default acceptable states satisfy two extra conditions: they do not justify literals defined in Δ_d in terms of literals defined in Δ_g, and the set of all literals in J is consistent. For an acceptable state, it suffices that the literals in J that are true in $\mathcal{I}$ and defined in Δ_d, are consistent. Since default acceptable states are acceptable states, Theorem 3.5 and Corollary 3.6 also hold for default acceptable states.

During standard model expansion, the main state-changing operations are to make $\mathcal{I}$ more precise (by making literals true, either through choice or propagation) and to make $\mathcal{I}$ less precise (by backjumping). When $S = \langle \Delta_\text{g}, \Delta_\text{d}, J, \mathcal{I}\rangle$ is a default acceptable state and model expansion modifies $\mathcal{I}$ into $\mathcal{I}$', the new state $\langle \Delta_\text{g}, \Delta_\text{d}, J, \mathcal{I}'\rangle$ is not necessarily a default acceptable state. The following propositions identify situations where acceptability is preserved.

Proposition 3.8. *Let $\langle \Delta_g, \Delta_d, J, \mathcal{I}\rangle$ be a default acceptable state, L a set of literals unknown in $\mathcal{I}$ and $\mathcal{I}'$ the consistent structure $\mathcal{I} \cup L$. If (**i**) literals of L either are not defined in Δ_d or have a direct justification in J and (**ii**) no direct justification in J contains the negation of a literal in L, then $\langle \Delta_g, \Delta_d, J, \mathcal{I}'\rangle$ is a default acceptable state.*

Proof. As the literals true in $\mathcal{I}$ and defined in Δ_d have a direct justification, it follows from (**i**) that all literals true in $\mathcal{I}$' and defined in Δ_d have a direct justification. As justifications in J are consistent with $\mathcal{I}$, then, by (**ii**), they are also consistent with $\mathcal{I}$'. Hence, J justifies all literals true in $\mathcal{I}$' and defined in Δ_d. □

Proposition 3.9. *Let $\langle \Delta_g, \Delta_d, J, \mathcal{I}\rangle$ be a default acceptable state. Then $\langle \Delta_g, \Delta_d, J, \mathcal{I}'\rangle$ with $\mathcal{I}' <_p \mathcal{I}$ is a default acceptable state.*

Proof. The justification J justifies all literals defined in Δ and true in $\mathcal{I}$. As $\mathcal{I}$' is a subset of $\mathcal{I}$, J justifies all literals defined in Δ and true in $\mathcal{I}$'. □

In a default acceptable state, literals defined in Δ_{g} are not allowed in direct justifications of literals defined in Δ_{d}. This restriction is quite limiting (see next section) but is to avoid hidden loops over Δ_{gd}. Such loops can only be detected by maintaining a justification over both Δ_{g} and Δ_{d}, which our current implementation does not do. Several methods exist to extend the class of default acceptable states. Literals l defined in Δ_{g} can be allowed in direct justifications of Δ_{d}, provided it can be established that l's justification cannot loop over Δ_{gd}. One case is when the body of the rule of l has no defined literals. A step further is to analyze the dependency graph: a literal defined in Δ_{g} can be allowed in the direct justification of a literal defined in Δ_{d} provided both literals do not belong to the same strongly connected component of the dependency graph. In that case, they cannot be part of the same cycle.

3.3 An Example

In the rest of the section, we illustrate the behavior of lazy model expansion on an artificial example, constructed in such a way that all main features are illustrated. In the next section, the processes involved are described in more detail.

We focus on the operation of the justification manager and its interaction with the solving process. The manager is activated in an unacceptable state, either when the solver falsifies a literal that occurs in a direct justification of J or when a true literal l defined in Δ_{d} is not justified by J. One option for repair is to search for a justification for l to extend J. In general this problem is as hard as the model expansion problem itself, as shown by Corollary 2.11. Our manager only searches *locally* for a direct justification that justifies l to extend J, and if it does not find one, it grounds l's definition and moves it to Δ_{g}.

Our example uses a theory $\mathcal{T}$ which states that a symmetric graph ($edge/2$) exists where at least one node other than the root node (predicate $root/1$) is reachable (predicate $R/1$) from the root node. The input structure $\mathcal{I}$ interprets the domain as $D = \{d_1, \ldots, d_n\}$ and the equality predicate as the identity relation on D (below omitted in $\mathcal{I}$). Predicates $edge, R$ and $root$ are not interpreted; R and $root$ are defined. In particular, $root$ is defined as the singleton $\{d_1\}$, specifying the root as d_1.

$$
\begin{array}{l}
P_{\mathcal{T}} \\
\left\{\begin{array}{rll}
P_{\mathcal{T}} & \leftarrow C_1 \land C_2 & (1) \\
C_1 & \leftarrow \exists x \in D : \neg root(x) \land R(x) & (2) \\
C_2 & \leftarrow \forall (x\ y) \in D^2 : edge(x,y) \Rightarrow edge(y,x) & (3) \\
\forall x \in D : root(x) & \leftarrow x = d_1 & (4) \\
\forall x \in D : R(x) & \leftarrow root(x) \lor \exists y \in D : edge(x,y) \land R(y) & (5)
\end{array}\right\}
\end{array}
$$

The lazy MX algorithm proceeds as follows:

1. The initial default acceptable state is $\langle \Delta_{\mathrm{g}}, \Delta_{\mathrm{d}}, J, \mathcal{I} \rangle$ in which Δ_{g}, $\mathcal{I}$ and J are empty, and $\Delta_{\mathrm{d}} = \Delta$.

2. Propagation over $\{P_{\mathcal{T}}, \Delta_{\mathrm{g}}\}$ sets $\mathcal{I}$ to $\{P_{\mathcal{T}}\}$. This expands the structure $\mathcal{I}$, but now the conditions of Proposition 3.8 are no longer satisfied. The resulting state is not

acceptable since $P_\mathcal{T}$ is true and defined in Δ_d while it has no direct justification in J. One option to repair acceptability is to extend J with a direct justification for $P_\mathcal{T}$. The atom $P_\mathcal{T}$ has a unique direct justification $\{C_1, C_2\}$ but extending J with it does not restore (default) acceptability since C_1, C_2 have no direct justification in J and $P_\mathcal{T}$ remains unjustified. Therefore, the alternative is taken and rule (1) is moved to Δ_g. Now, a default acceptable state is obtained.

3. Unit propagation sets $\mathcal{I}$ to $\{P_\mathcal{T}, C_1, C_2\}$. Now C_1 and C_2 have to be justified. Consider first C_2 and rule (3). As *edge* is open, our manager can build the direct justification $\{\neg edge(d, d') \mid (d, d') \in D^2\}$, that sets all negative *edge* literals true, and extends J with it (setting all positive *edge* literals true would be equally good). This justifies C_2 and avoids the grounding of the rule defining C_2.

4. Literal C_1 cannot be justified (with the local approach) since each of its direct justifications contains unjustified defined literals. However, as rule (2) is existentially quantified, one can avoid grounding the whole rule by performing a Tseitin transformation to isolate one instance and then only ground that instance. For the purpose of illustration, we make the (bad) choice of instantiating x with d_1:

$$\left\{ \begin{array}{lll} C_1 & \leftarrow (\neg root(d_1) \land R(d_1)) \lor T & (2a) \\ T & \leftarrow \exists x \in D \setminus \{d_1\} : \neg root(x) \land R(x) & (2b) \end{array} \right\}$$

Rule (2a) is moved to Δ_g and a default acceptable state is reached.

5. We are in an acceptable state in which no further propagation is possible, so a choice has to be made. As C_1 is true, the body of rule (2a) has to become true. Preferably not selecting a Tseitin (this would trigger more grounding), the first disjunct is selected by model expansion and propagation extends the structure with $\neg root(d_1)$ and $R(d_1)$. The literal $\neg root(d_1)$ is defined in Δ_d by rule (4) but cannot be justified since its unique direct justification $\{\neg(d_1 = d_1)\}$ is false. The manager partially grounds the definition of *root* and splits it up in a ground rule (4a) and a non-ground rule (4b) defining *root* for the other domain elements:

$$\left\{ \begin{array}{rll} root(d_1) & \leftarrow d_1 = d_1 & (4a) \\ \forall x \in D \setminus \{d_1\} : root(x) & \leftarrow x = d_1 & (4b) \end{array} \right\}$$

Rule (4a) is moved to Δ_g. Note that $root(d_1)$ is justified by $\{d_1 = d_1\}$ in Δ_gd, hence it is safe to use $root(d_1)$ in direct justifications in Δ_d. Whenever grounding has been done, the justification manager is interrupted by propagation, which can infer the truth of additional literals, or detect an inconsistency (which will result in backjumping). In both cases, the manager has to resume the revision of the justification afterwards, until an acceptable state is reached. Here, even though the resulting state is still unacceptable (due to the unjustified $R(d_1)$), the creation of the new rule (4a) in Δ_g interrupts the manager. Propagation using the new rule derives $root(d_1)$ and a conflict. After backtracking to $\mathcal{I} = \{P_\mathcal{T}, C_1, C_2\}$, the subsequent propagation sets the structure $\mathcal{I}$ to $\{P_\mathcal{T}, C_1, C_2, root(d_1), T\}$. Still not in a default acceptable state (T is not justified), rule (2b) is further transformed to split off another instance.

$$\left\{ \begin{array}{lll} T & \leftarrow (\neg root(d_2) \land R(d_2)) \lor T_2 & (2ba) \\ T_2 & \leftarrow \exists x \in D \setminus \{d_1, d_2\} : \neg root(x) \land R(x) & (2bb) \end{array} \right\}$$

Rule (2ba) is moved to Δ_g, while rule (2bb) remains in Δ_d. This state is default acceptable.

6. Again, the search avoids the new Tseitin T_2, choosing the first disjunct in rule (2ba) which propagates $\neg root(d_2)$ and $R(d_2)$. The literal $\neg root(d_2)$ is defined in Δ_d, but is justified by the direct justification $\{\neg(d_2 = d_1)\}$. The literal $R(d_2)$ cannot be justified by a direct justification (as all *edge* literals are false in the current justification graph) and rule (5) is transformed to split off the instance for d_2. Actually, this instance in turn has a disjunctive body with a complex subformula, so to avoid grounding the subformula, we break it up in two parts and introduce another Tseitin.
$$\left\{\begin{array}{rll} R(d_2) & \leftarrow root(d_2) \lor T_3 & (5aa) \\ T_3 & \leftarrow \exists y \in D : edge(d_2, y) \land R(y) & (5ab) \\ \forall x \in D \setminus \{d_2\} : R(x) & \leftarrow root(x) & \\ & \qquad \lor \exists y \in D : edge(x, y) \land R(y) & (5b) \end{array}\right\}$$
Rule (5aa) is moved to Δ_g, the others remain in Δ_d.

7. The current structure $\mathcal{I}$ is $\{P_{\mathcal{T}}, C_1, C_2, root(d_1), T, \neg root(d_2), R(d_2)\}$, hence propagation on rule (5aa) in Δ_g extends it with T_3. There is no direct justification justifying T_3 and, hence, rule (5ab) is partially grounded by splitting off the d_1 case:
$$\left\{\begin{array}{lll} T_3 & \leftarrow (edge(d_2, d_1) \land R(d_1)) \lor T_4 & (5aba) \\ T_4 & \leftarrow \exists y \in D \setminus \{d_1\} : edge(d_2, y) \land R(y) & (5abb) \end{array}\right\}$$
Rule (5aba) is moved to Δ_g while rule (5abb) remains in Δ_d.

8. The search selects the first disjunct of T_3's rule body and propagates $edge(d_2, d_1)$ and $R(d_1)$. The literal $R(d_1)$ is defined in Δ_d, but $\{root(d_1)\}$ is a direct justification for it. Extending J with this direct justification yields an acceptable but not default acceptable state, since $root(d_1)$ is defined in Δ_g. However, $root(d_1)$ is justified in Δ_{gd}, making it safe to extend J with this direct justification as discussed earlier. Now the justification manager faces a new problem: the true literal $edge(d_2, d_1)$ is in conflict with the direct justification $\{\neg edge(d, d') \mid (d, d') \in D^2\}$ of C_2 (rule (3)). To handle this conflict, it splits off the affected instance $(x = d_2, y = d_1)$ from this rule:
$$\left\{\begin{array}{lll} C_2 & \leftarrow (edge(d_2, d_1) \Rightarrow edge(d_1, d_2)) \land T_5 & (3a) \\ T_5 & \leftarrow \forall (x\ y) \in D^2 \setminus \{(d_2, d_1)\} : edge(x, y) \Rightarrow edge(y, x) & (3b) \end{array}\right\}$$
Rule (3a) is moved to Δ_g while rule (3b) remains in Δ_d. The direct justification of T_5 is set to $\{\neg edge(d, d') \mid (d, d') \in D^2 \setminus \{(d_2, d_1)\}\}$, the unaffected part of the direct justification of C_2. This restores acceptability.

9. Propagation on rule (3a) extends $\mathcal{I}$ with $edge(d_1, d_2)$ and T_5. The literal $edge(d_1, d_2)$, which is true, is in conflict with the direct justification for T_5 (rule (3b)). To resolve it, the justification manager partially grounds rule (3b) and splits off the instance $\{x = d_1, y = d_2\}$ as follows.
$$\left\{\begin{array}{lll} T_5 & \leftarrow (edge(d_1, d_2) \Rightarrow edge(d_2, d_1)) \land T_6 & (3ba) \\ T_6 & \leftarrow \forall (x\ y) \in D^2 \setminus \{(d_2, d_1), (d_1, d_2)\} : & \\ & \qquad edge(x, y) \Rightarrow edge(y, x) & (3bb) \end{array}\right\}$$

> Rule (3ba) is moved to Δ_g while rule (3bb) remains in Δ_d; T_6 inherits the direct justification of T_5 with $\neg edge(d_1, d_2)$ removed. Propagation on rule (3ba) extends $\mathcal{I}$ with T_6. The resulting state is acceptable, with T_6 defined in Δ_d but justified.

By now, Δ_g consists of the rules (1), (2a), (4a), (2ba), (5aa), (5aba), (3a), and (3ba), and the residual definition Δ_d consists of the rules (4b), (2bb), (5b), (5abb), and (3bb). The current structure $\mathcal{I}$ is $\{P_{\mathcal{T}}, C_1, C_2, root(d_1), \neg root(d_2), edge(d_2, d_1), edge(d_1, d_2), R(d_1), R(d_2), T, T_3, T_5, T_6\}$, a model of $P_{\mathcal{T}} \cup \Delta_g$.

Of these literals, $\neg root(d_2)$, $R(d_1)$ and T_6 are defined in Δ_d. Literal $\neg root(d_2)$, defined by rule (4b) has $\{\neg(d_2 = d_1)\}$ as direct justification. Literal $R(d_1)$, defined by rule (5b), has $\{root(d1)\}$ as direct justification. Literal T_6, defined by rule (3bb) has as direct justification the set of all negative *edge* literals over D except $edge(d_1, d_2)$ and $edge(d_2, d_1)$. To obtain a full model of the theory, $\mathcal{I}$ is extended with the literals of the above direct justifications. In this case, this assigns all open literals and the model can be completed by the well-founded model computation over Δ_{gd}. Actually, this can be done without grounding the definition (Jansen, Jorissen, & Janssens, 2013).

4. Justification Management

In Section 3.2, we have instantiated our general framework, developed in Section 3.1, for a justification manager that only has access to Δ_d. In the example of Section 3.3, the justification was constructed on demand, i.e., each time some literal needed a (different) direct justification, the body of its defining rule was analyzed and a justification was extracted. If that failed, part of the rule was grounded. This was called the *local approach*. One can also imagine a *global approach*, where more rules of Δ_d are considered at once in an attempt to select direct justifications that minimize the grounding of the rules as a whole. Obviously, a global approach will be more time consuming, so should not be applied every time an adjustment of the justification is required. In this section, we describe both approaches.

Before describing the algorithms, we introduce some notations and assume some normalizations have been done. The function nnf reduces a formula to its negation normal form. With S a set and s a single element, $S + s$ and $S - s$ are used as shorthands for $S \cup \{s\}$ and $S \backslash \{s\}$. With J a justification, we denote by $J[l \to d]$ the graph identical to J except that l is now justified by d. We assume quantifiers range over a single variable and variable names are not reused in a formula. Furthermore, we assume basic reductions have been applied to formulas, e.g., $\top \wedge \varphi$ reduces to φ, $\forall x \in \emptyset : \varphi$ reduces to $\mathbf{t}$, ...

4.1 The Local Approach

Algorithm 2 shows the top level of the lazy_mx model expansion algorithm, taking as input theory $\{P_{\mathcal{T}}, \Delta\}$ and input structure $\mathcal{I}_{in}$. Definitions Δ_d and Δ_g are initialized with Δ and the empty definition, respectively, and $\mathcal{I}$ is initialized as $\mathcal{I}_{in}$. The set of ground sentences T_g is initialized with the fact $P_{\mathcal{T}}$ and the initial justification J is empty. An auxiliary (FIFO) queue q_{ch} is initialized as empty. The latter keeps track of literals for which the direct justification needs to be checked.

The main loop performs model expansion over $T_g \cup \Delta_g$, interleaved with work by the justification manager towards establishing a default acceptable state. The model expansion part consists of propagation (the call to propagate), the test on whether the current state is inconsistent (with learning and backjumping), the test on whether a model of $T_g \cup \Delta_g$ has been found (returning the model and the justification) and of the choice step that selects a literal unknown in $T_g \cup \Delta_g$ and assigns it a value. Propagation returns literals that are entailed by a ground theory over a (partial) structure, for example by applying unit propagation and unfounded/wellfoundedness propagation (Mariën et al., 2008). The test for a model is only performed in a default acceptable state (i.e., when the queue q_{ch} is empty). If the test succeeds, this ensures that the well-founded model computation can expand the current structure $\mathcal{I}$ —extended with the direct justifications of all literals— into a model of the whole theory. Also the choice step only takes place in a default acceptable state; this ensures that the search space is limited to the state space of default acceptable states. The justification manager is activated when a propagation or choice step assigns a literal l. By calling check_literal, it is checked whether the current justification remains valid. If l is defined in Δ_d and has no justification, it needs a justification and is added to the queue q_{ch} for further processing by the justification manager. If $\neg l$ occurs in the justification of another literal l', the justification becomes inconsistent with $\mathcal{I}$ and l' needs another justification so it is also added to q_{ch}. The further processing is done by selecting elements from the queue and calling the lazy_ground function. The latter function first attempts to find a (different) consistent direct justification for l; if that fails, it splits off the rule instance defining l from Δ_d and partially grounds it, hence Δ_g is extended. The new clauses may trigger propagation; therefore the processing of queued literals is interleaved with propagation and, possibly, with backtracking . Note that backtracking might restore the consistency with $\mathcal{I}$ of the direct justification $J(l)$ of a literal l on q_{ch}.

4.1.1 Lazy Grounding of One Rule

The function lazy_ground, Algorithm 3, checks whether the literal l needs a direct justification; if not, it simply returns. Otherwise, it checks whether l has a valid justification, i.e., one that satisfies the invariants detailed below. If so, it also returns; otherwise, it passes the rule body that has to be used to construct a justification (the negation of the defining rule when the literal is negative) to build_djust, a function that attempts to find a valid direct justification. Besides the literal and the rule body, also an initial justification, derived from the rule body, is passed to build_djust. If the latter function is successful, the justification is updated and lazy_ground is done; if not, the direct justification of the literal l is set to *false* and split_and_ground is called to ground part of the rule defining l.

Before going into more details, we first analyze which properties we want to maintain in the current justification J. The direct justifications of literals in the q_{ch} queue are not considered part of J since they might be invalid. The global invariants of J are:

- no literals are unfounded in J (recall, negative cycles are allowed),
- the set of literals in J is consistent.

For each direct justification $S = J(l)$ of J for some l not on the queue, invariants of the lazy grounding process are:

- S contains no literals defined in Δ_g (unless such a literal is safely justified in Δ_{gd}, as discussed before),

Algorithm 2: The lazy_mx lazy model expansion algorithm.

Function lazy_mx (*atomic sentence* $P_{\mathcal{T}}$*, definition* Δ*, structure* $\mathcal{I}_{in}$)
  Output: either a model of Δ_{g} and J or *false*
  $T_{\mathrm{g}} := \{P_{\mathcal{T}}\}$; $\Delta_{\mathrm{g}} := \emptyset$; $\Delta_{\mathrm{d}} := \Delta$; $J := \emptyset$; $\mathcal{I} := \mathcal{I}_{in}$; $q_{ch} := \emptyset$;
  while *true* **do**
    $L :=$ propagate($T_{\mathrm{g}} \cup \Delta_{\mathrm{g}}, \mathcal{I}$);
    $\mathcal{I} := \mathcal{I} \cup L$;
    foreach $l \in L$ **do** q_{ch}:=check_literal(l,q_{ch});
    if $\mathcal{I}$ *is inconsistent* **then**
      T_{g} += learn_nogood($\mathcal{I}$, T_{g});
      if *conflict at root level* **then return** *false*;
      $\mathcal{I} := \mathcal{I}$ at state of backjump point;
    else if q_{ch} *is not empty* **then**
      $(l, q_{ch}) :=$ dequeue(q_{ch});
      lazy_ground(l);
    else if $\mathcal{I}$ *is a model of* $T_g \cup \Delta_g$ **then**
      return $\mathcal{I}$, J;
    else
      select choice literal l; $\mathcal{I} := \mathcal{I} + l$;
      q_{ch}:=check_literal(l,q_{ch});

Function check_literal (*literal* l*, literal queue* q_{ch})
  Data: global Δ_{d} and J **Output:** updated queue
  if l *defined in* Δ_d *and* $J(l) = $ *undef* **then** $q_{ch} :=$ enqueue(l,q_{ch}) ;
  foreach l' *such that* $\neg l \in J(l')$ **do** $q_{ch} :=$ enqueue(l',q_{ch}) ;
  return q_{ch};

Algorithm 3: The lazy grounding of a literal.

Function lazy_ground (*literal* l)
  Data: global Δ_{d}, J and $\mathcal{I}$
  if $l \in \mathcal{I}$ *and* l *defined in* Δ_d **then**
    if $J(l)$ *exists and obeys the invariants* **then return**;
    else
      $\varphi :=$ body of the rule defining l;
      if l *is a negative literal* **then** $\varphi :=$ nnf($\neg\varphi$) ;
      $dj :=$ build_djust($l, \varphi,$ init_just(l));
      if $dj \neq$ *false* **then** $J := J[l \to dj]$; **return**;
      else
        $J = J[l \to$ *false*$]$;
        split_and_ground(l);

- literals in S that are defined in Δ_{d} either have a direct justification in J or belong to the queue q_{ch}.

These invariants imply that a default acceptable state is reached when the q_{ch} queue is empty. Indeed, it follows from the invariants that the current justification is total in that situation and hence all literals that have a direct justification are justified (Definition 2.6). Due to the policy followed to queue literals, the current justification is also consistent with $\mathcal{I}$ while all literals true in $\mathcal{I}$ and defined in Δ_{d} have a justification, hence $\langle \Delta_{\mathrm{g}}, \Delta_{\mathrm{d}}, J, \mathcal{I} \rangle$ is a default acceptable state.

4.1.2 Building a Direct Justification

The purpose of build_djust, Algorithm 4, is to extend J with a suitable direct justification for literal l defined in Δ_{d}. Here l is a literal for which $J(l)$ is currently undefined or is inconsistent with $\mathcal{I}$. It is a recursive function which takes three parameters: (**i**) the literal l, (**ii**) the formula φ to be made true by the direct justification (initially the whole body of the rule defining the literal; note that the initialization takes the negation of the rule when the literal is negative), (**iii**) a description of the direct justification derived so far, initialized through init_just(l). For this algorithm, we assume φ is such that different quantifiers range over different variables.

Before going into details, we discuss how to represent direct justifications. Basically, we could represent a direct justification as a set of ground literals. However, this set can be quite large and using a ground representation might hence defy the purpose of lazy grounding. Instead, we represent a direct justification as a pair $\langle L, B \rangle$ with L a set of possibly non-ground literals and B a set of bindings $x_i \in D_i$ with x_i a variable and D_i a domain. A set of bindings $B = \{x_1 \in D_1, \ldots, x_n \in D_n\}$ represents the set of variable substitutions $S_B = \{\{x_1/d_1, \ldots, x_n/d_n\} \mid d_i \in D_i \text{ for each } i \in [1, n]\}$. The set of ground literals represented by $\langle L, B \rangle$ is then $\{l\theta \mid l \in L \text{ and } \theta \in S_B\}$. The direct justification of a literal $P(d_1, \ldots, d_n)$, defined by a rule $\forall \overline{x} \in \overline{D} : P(\overline{x}) \leftarrow \varphi$, is initialized by init_just(l) as $\langle \emptyset, \{x_1 \in \{d_1\}, \ldots, x_n \in \{d_n\}\} \rangle$. In effect, B allows to identify the relevant rule instantiation by providing the appropriate variable instantiation from the domains, while the set of literals is empty.

The build_djust algorithm searches for a set of literals making φ true. It works by recursively calling itself on subformulas of φ and composing the results afterwards into a larger justification for φ. When no such set of literals is found, for example because none exists that is consistent with all other direct justifications, *false* is returned.

The base case is when the formula is a literal. To make that literal true, all instances of the literal under the set of bindings B must be true, hence, the set of literals L is extended with the literal itself. The resulting direct justification has to satisfy all invariants, which is checked by the call to valid: it returns *true* for a call valid(l, dj) if dj satisfies the invariants to be (part of) a direct justification for l and $J[l \rightarrow dj]$ satisfies the invariants on the justification.

A universally quantified formula $\forall x \in D : \psi$ has to be true for each instance of the quantified variable. Hence, in the recursive call, the set of bindings B is extended with $x \in D$. For an existentially quantified formula, it suffices that one instance is true. Hence, a minimal approach is to try each instance separately until one succeeds; if all fail, *false* is

Algorithm 4: The build_djust algorithm.

Function build_djust (*literal l, formula φ and justification $\langle L, B\rangle$*)

  Input: B binds all free variables of φ
  Output: either a direct justification or *false*
  switch φ **do**
    case φ *is a literal*
      if *valid*$(l, \langle L \cup \{\varphi\}, B\rangle)$ **then return** $\langle L \cup \{\varphi\}, B\rangle$;
      else return *false*;
    case $\forall x \in D' : \psi$
      return *build_djust*$(l, \psi, \langle L, B \cup \{x \in D'\}\rangle)$;
    case $\exists x \in D' : \psi$
      if *large*(D') **then**
        return *build_djust*$(l, \psi, \langle L, B \cup \{x \in D'\}\rangle)$;
      foreach $d_i \in D'$ **do**
        $\langle L', B'\rangle :=$ build_djust$(l, \psi, \langle L, B \cup \{x \in \{d_i\}\}\rangle)$;
        if $\langle L', B'\rangle \neq$ *false* **then return** $\langle L', B'\rangle$;
      return *false*;
    case $\varphi_1 \wedge \ldots \wedge \varphi_n$
      foreach $i \in [1, n]$ **do**
        $\langle L', B'\rangle :=$ build_djust$(l, \varphi_i, \langle L, B\rangle)$;
        if $\langle L', B'\rangle =$ *false* **then return** *false*;
        else $\langle L, B\rangle := \langle L', B'\rangle$;
      return $\langle L, B\rangle$;
    case $\varphi_1 \vee \ldots \vee \varphi_n$
      foreach $i \in [1, n]$ **do**
        $\langle L', B'\rangle :=$ build_djust$(l, \varphi_i, \langle L, B\rangle)$;
        if $\langle L', B'\rangle \neq$ *false* **then return** $\langle L', B'\rangle$;
      return *false*;

returned. Note however that we do not want to iterate over each domain element if D is large, which would be similar to constructing the grounding itself. Instead, if D is large, we extend the binding with $x \in D$. Conjunction is similar to universal quantification, except that explicit iteration over each conjunct is needed. As soon as one conjunct fails, the whole conjunction fails. Disjunction is similar to existential quantification with a small domain.

Note that build_djust is non-deterministic due to choices for a domain element to justify an existentially quantified formula, or for a disjunct to justify a disjunction.

Example 4.1. Consider the following rule over a large domain D.

$$H \leftarrow \forall x \in D : \neg P(x) \vee (\exists y \in D : Q(x,y) \wedge \neg R(x,y))$$

Assume that J is empty and we have no loops to keep track of. Applying build_djust to H returns $\langle \{\neg P(x)\}, \{x \in D\} \rangle$ if the first disjunct $\neg P(x)$ in the body is chosen. This corresponds to the direct justification $\{\neg P(x) \mid x \in D\}$. Alternatively, if the second disjunct is chosen, it returns $\langle \{Q(x,y), \neg R(x,y)\}, \{x \in D, y \in D\} \rangle$, which represents the direct justification $\{Q(x,y) \mid x \in D, y \in D\} \cup \{\neg R(x,y) \mid x \in D, y \in D\}$.

4.1.3 Partially Grounding a Rule

The last bit of the lazy model expansion algorithm handles the case where no justification can be found and the definition of a literal l is to be grounded. A straightforward way would be to call one_step_ground on the rule defining l, and store the result in Δ_{g} and Δ_{d}. However, in many cases such an operation results in too much grounding.

Example 4.2. Consider a rule r_1 of the form $\forall x \in D : P(x) \leftarrow \varphi$ in a situation where no justification can be found for atom $P(d)$. Applying one_step_ground to r_1 would instantiate x with all elements in D, resulting in $|D|$ rules, while in fact it suffices to split r_1 in two rules, one for the instance $x = d$ and one for the remainder. Another example applies to a rule r_2 of the form $H \leftarrow \forall x \in D : Q(x) \vee R(x)$ and a direct justification $J(H) = \{Q(x) \mid x \in D\}$. When $Q(d)$ becomes false, the justification manager may need to ground this rule. Applying one_step_ground to it would instantiate the universally quantified x with all elements in D. Instead, it is better to split off the instance for $x = d$ and to introduce a Tseitin T for the remainder, producing $H \leftarrow (Q(d) \vee R(d)) \wedge T$ for Δ_{g} and $T \leftarrow \forall x \in D - d : Q(x) \vee R(x)$ for Δ_{d}. The direct justification for T can be obtained incrementally by removing $Q(d)$ from that of H, as discussed in Section 4.1.5.

The split_and_ground algorithm (Algorithm 5) has to ground part of the rule defining a given literal l, say $P(\overline{d})$. The first step is to split off the rule instance for which the rule defining l has to be grounded (the call to split). Let $\forall \overline{x} \in \overline{D} : P(\overline{x}) \leftarrow \varphi$ be the rule in Δ_{d} that defines $P(\overline{d})$. We then replace that rule by $\forall \overline{x} \in \overline{D} - \overline{d} : P(\overline{x}) \leftarrow \varphi$ in Δ_{d} and additionally return the rule $P(\overline{d}) \leftarrow \varphi[\overline{x}/\overline{d}]$. Afterwards, we apply one_step_ground to the latter rule and add the computed rules to either Δ_{g} or Δ_{d}. [2]

The result of split_and_ground is that definition Δ_{gd} is "more" ground than the previous one. The limit is a ground definition Δ_{gd} in which Δ_{d} is empty and Δ_{g} is strongly $voc(\Delta)$-equivalent with Δ.

2. Recall, the head of a new grounded rule is always different from the head of already grounded rules.

Algorithm 5: The split_and_ground algorithm.

Function split_and_ground (*literal* l)
  Input: l is defined in Δ_d
  Result: update to Δ_g, Δ_d, J, and q_{ch}
  $r :=$ split(l); // split updates Δ_d
  $(\Delta'_\mathrm{g}, \Delta'_\mathrm{d}) :=$ one_step_ground(r);
  $\Delta_\mathrm{g} \cup = \Delta_\mathrm{g}$'; $\Delta_\mathrm{d} \cup = \Delta_\mathrm{d}$';

Even if no justification was found, we can do better than just splitting off l and applying one_step_ground, as shown in Example 4.2. First, splitting can be made significantly more intelligent, which is discussed in Section 4.1.5. Second, we can improve one_step_ground to only ground part of expressions if possible, which we describe below.

Improving one_step_ground. Applying one_step_ground to a rule $l \leftarrow \varphi$ iterates over all subformulas/instantiations of φ. For example if φ is the sentence $\exists x \in D : P(x)$, the result consists of $|D|$ new rules and as many new Tseitin symbols. Instead, depending on the value of l, it is sufficient to introduce only one (or some) of these subformulas, as shown in Algorithm 6, which extends the switch statement of one_step_ground with two higher-priority cases. If l is true, it is sufficient to ground one disjunct/existential instantiation and to delay the rest by Tseitin introduction. If l is false, we take a similar approach for conjunction/universal quantification.

Algorithm 6: Additional cases for the one_step_ground algorithm.

switch r **do**
  case $l \leftarrow \varphi_1 \vee \ldots \vee \varphi_n$ *and* $\mathcal{I}(l) = \mathbf{t}$
    choose $i \in [1, n]$;
    return $\langle \{l \leftarrow \varphi_i \vee T\}, \{T \leftarrow \bigvee_{j \in \{1 \ldots n\} - i} \varphi_j\} \rangle$;
  case $l \leftarrow \exists x \in D : \varphi$ *and* $\mathcal{I}(l) = \mathbf{t}$
    choose $d \in D$;
    return $\langle \{l \leftarrow \varphi[x/d] \vee T\}, \{T \leftarrow \exists x \in D - d : \varphi\} \rangle$;
  analogous cases for $\wedge$ and $\forall$ in combination with $\mathcal{I}(l) = \mathbf{f}$.

4.1.4 Algorithmic Properties

Correctness and termination of the presented algorithms is discussed in the following theorem.

Theorem 4.3 (Correctness and termination)**.** *If lazy_mx returns an interpretation $\mathcal{I}$, then expanding $\mathcal{I}$ with the literals in the direct justifications of J, applying the well-founded evaluation over Δ_{gd} and restricting it to voc$(\mathcal{T})$ results in a model of $\mathcal{T}$. If the algorithm returns false, no interpretation exists that is more precise than $\mathcal{I}_{in}$ and satisfies $\mathcal{T}$.*

Algorithm lazy_mx *terminates if $\mathcal{I}$ is over a finite domain D. Otherwise, termination is possible but not guaranteed.*[3]

Proof. If lazy_mx returns an interpretation $\mathcal{I}$, $\mathcal{I}$ is a model of Δ_{g} and q_{ch} is empty. Given the properties of split_and_ground, after applying lazy_ground for a literal l, either l has a valid justification or is defined in Δ_{g}. Hence if q_{ch} is empty, we are in a default acceptable state and, by Theorem 3.5, $\mathcal{I}$ can be expanded into a model of the whole theory. If lazy_mx returns *false*, it has been proven that Δ_{g} has no models in $\mathcal{I}_{in}$. In that case, there can also be no models of Δ_{gd} and hence $\mathcal{T}$ also has no models expanding $\mathcal{I}_{in}$.

Without calls to lazy_ground, the search algorithm terminates for any finite Δ_{g}; the function lazy_ground itself produces an ever-increasing ground theory Δ_{g} with the full grounding as limit. Hence, lazy_mx always terminates if D is finite. If D is infinite, the limit of Δ_{g} is an infinite grounding, so termination cannot be guaranteed. □

4.1.5 Symbolic Justifications, Incremental Querying and Splitting

The algorithms presented above are sound and complete. However, they can be further improved by taking the formulas from which justifications are derived into account.

Symbolic justifications and incremental querying. When multiple justifications exist for (subformulas of) a formula φ, grounding can be further delayed.

Example 4.4. Consider the formula $\forall x \in D : P(x) \lor Q(x)$, where both $\langle \{P(x)\}, \{x \in D\} \rangle$ and $\langle \{Q(x)\}, \{x \in D\} \rangle$ are justifications. From that, we could derive the justification: for each $d \in D$, make either $P(d)$ *or* $Q(d)$ true. Hence, more grounding is necessary *only* when both $P(d)$ and $Q(d)$ become false for the same d.

We can do this automatically by changing build_djust as follows:

- The algorithm is allowed to select multiple disjunctions / existential quantifications even if a valid justification was already found for one (Lines 13 and 24).
- build_djust no longer returns a justification, but a symbolic *justification formula* that entails the original formula. The formula is built during build_djust and reflects the subformulas/instantiations that have been selected. From ψ, the justification itself can be derived directly as the set of non-false literals in the (full) grounding of ψ. For example, for a formula $\forall x \in D : P(x) \lor Q(x)$, instead of the justification $\{P(x) \mid x \in D\}$, build_djust might now return the justification formula $\forall x \in D : P(x) \lor Q(x)$.
- The validity check (valid) is extended to return false if the justification formula is false.

By allowing more complex formulas (instead of a conjunction of universally quantified literals), the validity check —whether a formula has become false after incremental changes to $\mathcal{I}$— is now more expensive. This is in fact an *incremental query* problem. In the experiments, we limit the depth of the allowed formulas and use a straightforward (expensive) algorithm that evaluates the whole formula whenever an assignment can falsify it.

3. It is possible to change the integration of lazy_ground in lazy_mx to guarantee termination if a finite model exists, see Section 5.1.

Body splitting. As described in Algorithm 3 of Section 4.1.3, split simply splits off the rule instance that defines l, then one_step_ground grounds this rule instance step by step, in accordance with the structure of the formula. However, when the grounding is triggered by a conflict with the current justification, one_step_ground is blind for the origin of the conflict. By using the conflicting literals, one could focus on grounding the part of the formula that contributes to the conflict. One can do so by restructuring the rule in a part to be grounded, that contains the conflict, and a part not to be grounded, for which the old justification can be adjusted to still apply. The latter part can then be split off by introducing new Tseitins and the transformation is called body splitting. This approach can be inserted in Algorithm 3 after the call to split. For this, the original justification (call it j_{old}) is passed as an extra argument to split_and_ground.

Example 4.5. Consider the rule $h \leftarrow \forall x \in D : P(x)$; let $\langle\{P(x)\}, \{x \in D\}\rangle$ be the justification for h true in $\mathcal{I}$. When $P(d)$ becomes false, it is easy to see that we can split off the "violating instantiation" by rewriting the original rule into $h \leftarrow P(d) \land T$ and adding the rule $T \leftarrow \forall x \in D - d : P(x)$. Crucially, a justification for the second part can be derived from the original justification, namely $\langle\{P(x)\}, \{x \in D - d\}\rangle$. The second part can hence be added to $\Delta_{\rm d}$ and its justification to J while the first part is added to $\Delta_{\rm g}$.

This revision of a rule r and its direct justification j_{old} can be done in an efficient way. Assume that v is a true domain literal in the partial structure and that the direct justification of rule r contains its negation $\neg v$. The implementation is such that the binding(s) for which the justification instantiates to $\neg v$ can be extracted from the representation of the direct justification of the rule. For simplicity, assume $\{\overline{x} = d_1, \ldots, d_n\}$ is the single such instance. A recursive algorithm visits the formula φ in the body of r depth-first. Whenever a quantification $\forall x \in D : \varphi$ is encountered with x equal to $x_j \in \overline{x}$, it is replaced by $(\forall x \in D - d_j : \varphi) \land \varphi[x = d_j]$. Then a Tseitin transformation is applied to the left-hand conjunct and the algorithm recurses on the right-hand conjunct with what remains of the binding. The new rule defining the new Tseitin has $j_{old} - v$ as a direct justification. Similarly, an existential quantification is replaced by a disjunction. The result is a set of new rules for which no new justification has to be sought and a smaller rule r' which is passed to one_step_ground. Correctness follows from the fact the $j_{old} - v$ is a valid justification, none of the new rules contains v, and from the correctness of the Tseitin transformation.

Example 4.6. In Example 4.1, justifications were sought for H in the rule

$$H \leftarrow \forall x \in D : \neg P(x) \lor (\exists y \in D : Q(x, y) \land \neg R(x, y)).$$

Assume we selected the justification $J = \{Q(x, y) \mid x \in D, y \in D\} \cup \{\neg R(x, y) \mid x \in D, y \in D\}$. When $P(d_1)$ is true in $\mathcal{I}$, and $l = Q(d_1, d_2)$ becomes false, J is no longer consistent with $\mathcal{I}$ and cannot be repaired. $J - l$, however, is still consistent with $\mathcal{I}$, but is not a justification of the whole body. On the other hand, $J - l$ is a justification for the subformula $\neg P(x) \lor \exists y \in D : Q(x, y) \land \neg R(x, y)$ for each instantiation of x different from d_1. Consequently, we can split the quantification $\forall x \in D :$ into $x \in D - d_1$ and $x = d_1$ and apply a Tseitin transformation to the former. Afterwards, we recursively visit the latter formula and apply a similar reasoning to the existential quantification. The operations on

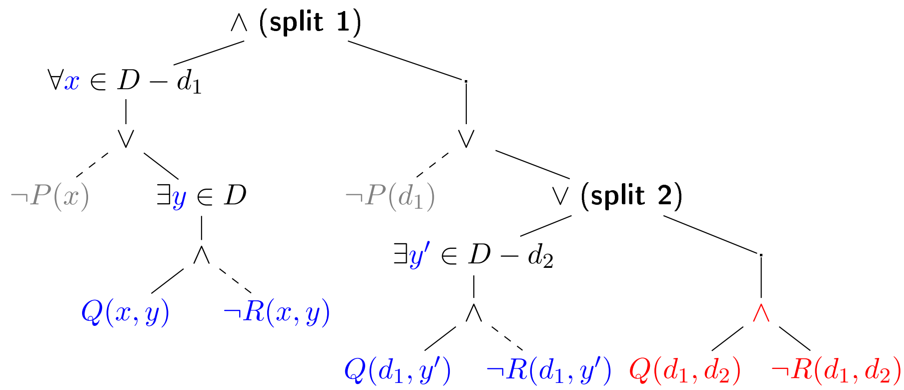

Figure 2: The rule body $\forall x \in D : \neg P(x) \lor \exists y \in D : Q(x, y) \land \neg R(x, y)$ is split for a violating literal $Q(d_1, d_2)$. The original justification without $Q(d_1, d_2)$ is a justification of the left-hand side of all splits, with the justification formula shown in blue. The remaining non-justified formula is shown in red.

the formula are illustrated in Figure 2. The result consists of the following rules, where the rule for H is now even ground.

$$\left\{ \begin{array}{l} H \leftarrow T_1 \land (\neg P(d_1) \lor (T_2 \lor (Q(d_1, d_2) \land \neg R(d_1, d_2)))). \\ T_1 \leftarrow \forall x \in D - d_1 : \neg P(x) \lor \exists y \in D : Q(x, y) \land \neg R(x, y). \\ T_2 \leftarrow \exists y \in D - d_2 : Q(d_1, y) \land \neg R(d_1, y). \end{array} \right\}$$

To further optimize the traversal of the formula φ, build_djust can be extended to store the path taken through the parse tree of φ and the direct justifications of the subformulas.

Example 4.7. Assume the rule $C_1 \leftarrow \forall (x\ y) \in D^2 : \neg edge(x, y) \lor edge(y, x)$ has to be justified, J is empty and $\mathcal{I}$ does not interpret $edge$. The build_djust algorithm recursively visits the body of the rule until $\neg edge(x, y)$ is returned as it is a valid literal to use. Going up one level, we store that for $\neg edge(x, y) \lor edge(y, x)$, we selected $\{\neg edge(x, y)\}$. Assuming no more disjuncts are selected, $\neg edge(x, y)$ is returned again. Going back up through both quantifications, we store that, for both quantifications, we selected D as the set of relevant domain elements, and build_djust returns the justification formula $\forall (x\ y) \in D^2 : \neg edge(x, y)$.

If build_djust is given access to j_{old}, similar optimizations are possible for repairing a direct justification. Consider again Example 4.6, but assume $P(d_1)$ is unknown in $\mathcal{I}$. In this case, the left branch of Figure 2 can also be transformed in a rule with a still valid, direct justification. For the right branch, a repair is to select the disjunct $\neg P(d_1)$ as direct justification for the rule $H \leftarrow T_1 \land (\neg P(d_1) \lor (\exists y \in (D - d_1) : Q(d_1, y) \land \neg R(d_1, y)))$, where T_1 is as in Example 4.6.

4.2 The Global Approach

Finding justifications using the greedy local approach can easily lead to more grounding than necessary. Consider for example the sentences $\forall x \in D : P(x)$ and $\forall x \in D : P(x) \Rightarrow Q(x)$.

Applying the local approach to the second sentence first (with an empty J), might result in the construction which makes atoms over P false. Then applying the local approach to the first sentence finds no valid justification for it; so it has to be fully grounded. The global approach takes a set of rules as input and tries to select direct justifications for them such that the *expected* grounding size of the whole set is minimal.

We cast the task, called the *optimal justification* problem, as a problem on a graph as follows. The graph consists of two types of nodes, *rule nodes* R and *justification nodes* J. A justification node is a symbolic set of literals representing a possible justification (for the literals defined by a rule in Δ_{d}, given the current partial structure $\mathcal{I}$). A rule node is a pair $\langle r, t\rangle$ of a rule r in Δ_{d} and a truth value t ($\mathbf{t}$, $\mathbf{f}$, $\mathbf{u}$); A pair $\langle r, \mathbf{t}\rangle$ for a rule r with head l expresses that there exists a direct justification for l; the pair $\langle r, \mathbf{f}\rangle$ that there exists a direct justification for $\neg l$ and the pair $\langle r, \mathbf{u}\rangle$ that r has no justification.

There are three types of edges:

- *Valid edges* between a rule node $\langle r, \mathbf{t}\rangle$ ($\langle r, \mathbf{f}\rangle$) and a justification node j where j justifies (the negation of) the head of r.
- *Conflict edges* between (**i**) rule nodes of the same rule with different truth value, (**ii**) justification nodes that contain opposite literals, (**iii**) a rule node $\langle r, \mathbf{t}\rangle$ ($\langle r, \mathbf{f}\rangle$) where r defines l, and a justification node that contains $\neg l$ (l), and (**iv**) a rule node $\langle r, \mathbf{u}\rangle$, where r defines l, and a justification node that contains l or $\neg l$ (a conflict because l (or $\neg l$) needs a justification).
- *Depends-on edges* between a justification node j and a rule node $\langle r, \mathbf{t}\rangle$ ($\langle r, \mathbf{f}\rangle$) where j contains negative (positive) literals defined by r.

The aim is then to select subsets $R_{sel} \subseteq R$ and $J_{sel} \subseteq J$ such that

- Each selected rule node is connected with a valid edge to at least one selected justification node.
- No conflict edges exist between pairs of selected nodes.
- Neither positive nor mixed cycles exist in the subgraph consisting of the valid and depends-on edges between the selected nodes.

From a selection $\{R_{sel}, J_{sel}\}$ satisfying these constraints, an initial justification J can be extracted as follows. A literal l ($\neg l$) is given a direct justification if it is defined by a rule r such that $\langle r, \mathbf{t}\rangle$ ($\langle r, \mathbf{f}\rangle$) is a selected rule. Its direct justification is the union of the justifications of the justification nodes in J_{sel} connected with $\langle r, \mathbf{t}\rangle$ ($\langle r, \mathbf{f}\rangle$) through a valid edge. Moreover, all literals defined by rules for which no rule node was selected can be added to the initial q_{ch} queue, to be handled by the local approach, as a complete solution must have a justification for them. When $\langle r, \mathbf{u}\rangle$ is selected, it means that the grounding of instances of the rule is delayed until the literals it defines become assigned.

This type of problem is somewhat related to the NP-hard *hitting set* (or *set cover*) problem (Karp, 1972): given a set of "top" and "bottom" nodes and edges between them, the task is to find a minimal set of bottom nodes such that each top node has an edge to at least one selected bottom node.

Given a default acceptable state $\langle \Delta_g, \Delta_d, J, \mathcal{I} \rangle$, the input for the optimal justification problem is generated as follows. For any rule in Δ_d, a node is constructed for each of the three truth values (only one when the truth value is known in $\mathcal{I}$) and also their conflict edges are added. Valid edges and justification nodes are obtained using a (straightforward) adaptation of build_djust that returns a set of possible justifications that make the head of a rule true (false). E.g., for a rule $\forall \overline{x} \in \overline{D} : P(\overline{x}) \leftarrow \varphi$, build_djust is called with the literal $P(\overline{x})$ and the binding is initialized at $\{\overline{x} \in \overline{D}\}$. Conflict and depends-on edges are derived by checking dependencies between justifications and between rules and justifications. To keep this efficient, it is done on the symbolic level.

Example 4.8. Consider the theory of our running example, after $P_{\mathcal{T}}$, C_1 and C_2 have been propagated to be true. Definition Δ_d is then

$$\left\{ \begin{array}{rll} C_1 & \leftarrow \exists x \in D : \neg root(x) \wedge R(x) & (2) \\ C_2 & \leftarrow \forall (x\ y) \in D^2 : edge(x,y) \Rightarrow edge(y,x) & (3) \\ \forall x \in D : root(x) & \leftarrow x = d_1 & (4) \\ \forall x \in D : R(x) & \leftarrow root(x) \vee \exists y \in D : edge(x,y) \wedge R(y) & (5) \end{array} \right\}$$

The associated optimal construction set input is shown in Figure 3. Note that in rule nodes, we use the defined head literals to identify the rule. Literal C_1 and C_2 are true in $\mathcal{I}$, hence there is only one rule node for rules (2) and (3). Neither $root(x)$ nor $\neg root(x)$ can be justified for all $x \in D$, hence there is only a $\langle root, \mathbf{u} \rangle$ tuple.

There are four solutions that are subset-maximal with respect to rule nodes, namely the following rule node selections:

$$\{\langle R, \mathbf{u} \rangle, \langle root, \mathbf{u} \rangle, \langle C_2, \mathbf{t} \rangle\} \quad \text{(a)}$$

$$\{\langle R, \mathbf{f} \rangle, \langle C_2, \mathbf{t} \rangle\} \quad \text{(b)}$$

$$\{\langle R, \mathbf{t} \rangle, \langle C_2, \mathbf{t} \rangle\} \quad \text{(c)}$$

$$\{\langle C_1, \mathbf{t} \rangle, \langle C_2, \mathbf{t} \rangle\} \quad \text{(d)}$$

For each of these, multiple justification selections are possible (also shown in Figure 3). For C_1, we have to select justification (iv), but for C_2 we can choose from (v) or (vi) (but not both).

The objective is then not to maximize the number of selected rule nodes, but to minimize the expected grounding size. To obtain an estimate of the expected grounding size, the following conditions were taken into account:

- It should depend on the size of the grounding of the rule.
- Assigning multiple justifications to a rule should result in a lower estimate as the rule will only be grounded when all are false.
- Variables occurring in multiple justifications result in less matching instantiations.
- In most practical applications, the number of false atoms far exceeds the number of true ones in any model. Hence, positive literals in a justification should have a higher cost than negative ones.

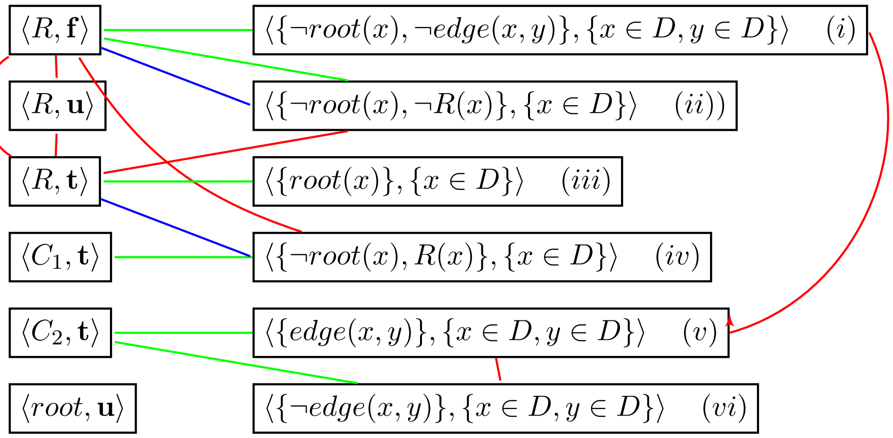

Figure 3: The graph that is part of the input to the optimal justification problem of Example 4.8. Rule nodes are shown on the left, justification nodes on the right; valid edges are shown in green, conflict edges in red and depends-on edges in blue. For readability, conflicts between justifications and unknown rule nodes are not shown.

We approximate the expected grounding size with the function exp_{size} which takes as input a rule r (with head h), the selected type of justification (rule not selected ($\mathbf{n}$), no justification ($\mathbf{u}$), a justification for h ($\mathbf{t}$) or a justification for $\neg h$ ($\mathbf{f}$)) and the set of justifications J. The function returns the expected grounding size of the rule ($size(r)$, defined below) weighted depending on the type of justification. The weights are derived from two estimates: p_{val} is the probability an atom will become assigned and p_{tr} is the probability of an assigned atom to be true. Hence, as defined formally below, for non-delayed rules ($\mathbf{n}$), the full size is used; a rule without justification ($\mathbf{u}$) is weighted by p_{val}; a true justification ($\mathbf{t}$) is weighted by p_{tr}, a false one ($\mathbf{f}$) by $1 - p_{tr}$; the latter two weights are multiplied by a product over the justifications j in J. Each factor of this product is a sum of two terms, namely $p_{\mathbf{t}}$ times the number of negative literals in j and $1 - p_{tr}$ times the number of positive literals in j. The effect is that the expected size decreases as the number of justifications increases and that the expected size increases as a justification has more literals.

$$
\begin{aligned}
exp_{size}(r, \mathbf{n}, \emptyset) &= size(r) \\
exp_{size}(r, \mathbf{u}, \emptyset) &= size(r) \times p_{val} \\
exp_{size}(r, \mathbf{f}, J) &= size(r) \times p_{val} \times (1 - p_{tr}) \times \prod_{j \in J} ((1 - p_{tr}) \times |\text{pos. lits.} \in j| \\
&\qquad\qquad\qquad\qquad\qquad\qquad + p_{tr} \times |\text{neg. lits.} \in j|) \\
exp_{size}(r, \mathbf{t}, J) &= size(r) \times p_{val} \times p_{tr} \times \prod_{j \in J} ((1 - p_{tr}) \times |\text{pos. lits.} \in j| \\
&\qquad\qquad\qquad\qquad\qquad\qquad + p_{tr} \times |\text{neg. lits.} \in j|)
\end{aligned}
$$

For the probabilities, we assumed p_{val} to be small (currently 0.1) to reflect the hope that lots of literals will not get a value, and p_{tr} is less than half, to reflect that atoms are more often assigned false than true. The function *size* is defined below. The function returns the number of atoms in the grounding of a rule or formula, except for existential quantification and disjunction. For these, we take into account that they can be grounded only partially using Tseitin transformation, by taking the logarithm of the total grounding size.

$$\begin{aligned}
size(L) &= 1 \\
size(L \leftarrow \varphi) &= size(\varphi) + 1 \\
size(\forall x \in D : \varphi) &= D \times size(\varphi) \\
size(\phi_1 \land \ldots \land \varphi_n) &= \sum_{i \in [1,n]} size(\varphi_i) \\
size(\exists x \in D : \varphi) &= log(D) \times size(\varphi) \\
size(\phi_1 \lor \ldots \lor \varphi_n) &= log(n) \times \frac{\sum_{i \in [1,n]} size(\varphi_i)}{n}
\end{aligned}$$

Solutions to the optimal justification problem should minimize the term

$$\sum_{r \in \Delta_{\mathrm{d}}} exp_{size}(r, t(r), J(r))$$

with $t(r)$ the type ($\mathbf{t}$, $\mathbf{f}$, or $\mathbf{u}$) and $J(r)$ the justification of the literal defined by r.

Example 4.9 (Continued from Example 4.8)**.** The size of rule C_1 is $1+log(D)\times 2$, that of C_2 is $1+D^2\times log(2)$, that of *root* is $D\times(1+1)$, and that of R is $D\times(1+log(2)\times(1+log(D)\times 2)/2$. Consider assigning justification (iv) to C_1: this results in an expected cost for that rule of $(1 + log(D) \times 2) \times 0.3 \times 1$ (as the construction relies on making R true). Additionally, it would force the grounding of the rule defining R, increasing the cost with the size of the rule for R. The optimal solution for the problem in Figure 3 is then rule node selection (a) with justification (vi) for C_2. Its cost is the sum of $(1 + D^2 \times log(2) \times 0.3$ (for justification (vi)) and $1 + log(D) \times 2$ (the expected size of the rule for C_1). Now, only the rule for C_1 has to be passed to the local approach.

To solve the optimal justification problem, IDP's optimization inference itself is applied to a (meta-level) declarative specification of the task.[4] For larger theories $\mathcal{T}$, the problem turns out to be quite hard, so two approximations were considered to reduce the search space. First, the number of selected justifications for any rule was limited to 2. Second, as the values of $size(r)$ can grow quite large, the approximation $\lceil \log(size(r)) \rceil$ is used (a standard approach). Rounding to integer values was applied as IDP's support for floating point number is still preliminary. The resulting specification could be solved to optimality within seconds for all tested theories. During lazy model expansion, the global approach is applied in the initial phase when all Tseitin literals representing sentences in the original theory have been propagated true.

5. Heuristics and Inference Tasks

This section discusses how to tune the lazy grounding and the search heuristics of the underlying SAT solver to obtain an effective implementation of lazy model expansion. We

4. The specification is part of IDP's public distribution.

also describe a few other inferences tasks beyond model expansion that are useful in the context of lazy grounding. A few less important issues are discussed in Appendix A.

5.1 Heuristics

Heuristics play an important role in the lazy grounding algorithms, as they serve to find the right balance between how much to ground and how long to search. We first discuss how our heuristics were chosen. Afterwards, we discuss an alternative approach to minimize grounding.

5.1.1 The Balance between Grounding and Search

The algorithms leave room for a number of heuristic choices that can have an important effect on the performance. We now briefly discuss these choices. As a guideline for our decisions, the following principles are used:

- Avoid leaving the search process without enough information to make an informed decision; for example, avoid losing too much (unit) propagation or introducing too many Tseitin symbols.
- Prevent creating a grounding that is too large; this may for example happen as the result of a very long propagate-ground sequence.

Recall, the goal is not to create a minimal grounding, but to solve model expansion problems while avoiding a too large grounding.

Below, we introduce a number of parameters that affect these heuristics. The exact values used in the experimental evaluation for the parameters introduced below are specified in Appendix A.

In split_and_ground, when handling a disjunction or existential quantification, there is a choice on how many disjuncts to expand. If we expand one instantiation at a time for a rule $h \leftarrow \exists x \in D : P(x)$, as done in Algorithm 6 (lines 3 and 6), iterative application results in a ground theory

$$\begin{aligned} h &\leftarrow P(d_1) \lor T_1 \\ T_1 &\leftarrow P(d_2) \lor T_2 \\ T_2 &\leftarrow P(d_3) \lor T_3 \\ &\vdots \\ T_n &\leftarrow \exists x \in D \setminus \{d_1, d_2, \ldots, d_n\} : P(x). \end{aligned}$$

A SAT-solver such as MiniSAT, which is used in the IDP system, initially assigns **f** to the $P(d_i)$ atoms; such a choice triggers an iteration of propagation and grounding. The resulting thrashing behavior can be reduced somewhat, and the grounding is more compact when the grounding introduces n disjuncts at a time:

$$\begin{aligned} h &\leftarrow P(d_1) \lor \ldots \lor P(d_n) \lor T \\ T &\leftarrow \exists x \in D \setminus \{d_1, d_2, \ldots, d_n\} : P(x). \end{aligned}$$

To further remedy this, two search-related heuristics are changed. First, the initial truth value is randomized, but favoring false (as in most models, many more atoms are false than true). Second, search algorithms typically *restart* after an (ever-increasing) threshold on the number of conflicts, sometimes caching the truth value assigned to atoms (*polarity caching*). This allows the solver to take learned information into account in the search heuristic while staying approximately in the same part of the search space. In case of lazy grounding, we might want to jump to another part of the search space when we come across long propagate-ground sequences. To this end, we introduce the concept of *randomized restarts*, which take place after an (ever-increasing) threshold on the number of times Δ_g is extended and randomly flipping some of the cached truth values.

In addition, build_djust always returns *false* if it is estimated that the formula has a small grounding. Indeed, grounding such formulas can help the search. Whether a formula is considered small is determined in terms of its (estimated) grounding size. The same strategy is used in split_and_ground: whenever the formula to which one_step_ground would be applied is small, ground is applied instead, to completely ground the formula.

5.1.2 Late Grounding

Grounding is applied during the search process as soon as unit propagation has taken place. The result is a focus on the current location in the search space, but with the danger of grounding too much if there is no solution in that part of the space. Alternatively, we could apply the opposite strategy, namely to ground as late as possible: only apply additional grounding when the search algorithm terminates without ever having found a model in an acceptable default state. Such a strategy is well-known from the fields of incremental proving and planning, where the domain (number of time steps) is only increased after search over the previous, smaller bound has finished. This guarantees a minimal grounding. A prototype of this strategy has been implemented in IDP with good results on planning problems.

5.2 Related Inference Tasks

The bulk of the paper focuses on model expansion (MX) for FO(ID) theories $\mathcal{T}$, for which solutions are structures that are two-valued on $voc(\mathcal{T})$. Often, one is only interested in a small subset of the symbols in $voc(\mathcal{T})$. This is for example the case for model generation for $\exists$SO(ID), the language which extends FO(ID) with existential quantification over relations. An $\exists$SO(ID) problem $\exists P_1, \dots, P_n : \mathcal{T}$ with an initial structure $\mathcal{I}$, relation symbols $P_1, \dots, P_n$, and $\mathcal{T}$ an FO(ID) theory, can be solved by model generation for the FO(ID) theory $\mathcal{T}$ with initial structure $\mathcal{I}$ and by dropping the interpretation of the symbols $P_1, \dots, P_n$ in the models. Another example is query evaluation for FO(ID): given a theory $\mathcal{T}$, an initial structure $\mathcal{I}$ and a formula φ with free variables $\overline{x}$ (all in FO(ID)), the purpose of evaluating the query $\langle \mathcal{T}, \mathcal{I}, \varphi \rangle$ is to find assignments of domain elements $\overline{d}$ to $\overline{x}$ such that a model of $\mathcal{T}$ exists that expands $\mathcal{I}$ and in which $\varphi[\overline{x}/\overline{d}]$ is true. To solve it by model expansion in FO(ID), a new predicate symbol T is introduced and answers to the query are tuples of domain elements $\overline{d}$ such that $T(\overline{d})$ is true in a model of the theory $\mathcal{T}$ extended with the sentence $\exists \overline{x} \in \overline{D} : T(\overline{x})$ and the definition $\{\forall \overline{x} \in \overline{D} : T(\overline{x}) \leftarrow \varphi\}$.

In both cases, approaches using (standard) model expansion compute a total interpretation and afterwards drop all unnecessary information, which is quite inefficient. Lazy model expansion can save a lot of work by only partially grounding the theory. However, once a model is found for the grounded part, the justifications and the remaining definitions are used to expand the structure to a model of the full theory. Although this expansion is obtained in polynomial time, it is still inefficient when afterwards a large part of the model is dropped.

To remedy this, we define a variant of the model expansion task, denoted *restricted* MX. Restricted MX takes as input a theory $\mathcal{T}$, a structure $\mathcal{I}$ and an additional list of symbols O, called *output symbols*.[5] Solutions are then structures M which are two-valued on all symbols in O and for which an expansion exists that extends $\mathcal{I}$ and is a model of $\mathcal{T}$. Adapting lazy grounding to solve restricted MX can be done through an analysis of which justifications need not be added (completely) to the structure, splitting Δ_{gd} into multiple definitions and only evaluating those defining output symbols or symbols on which those depend (using a stratification argument).

The above-mentioned inference tasks can be cast trivially to restricted MX problems and lazy restricted MX then greatly improves the efficiency with respect to ground-and-solve, as shown in the experimental section.

The extension of FO(ID) with *procedurally interpreted* symbols (De Cat et al., 2014) provides another class of interesting problems. Such predicate symbols have a fixed interpretation, but to know whether a tuple belongs to the predicate, a procedural function has to be executed. Such an approach provides a clean way to combine declarative and procedural specifications. Consider for example a symbol $isPrime(\mathbb{N})$ that is interpreted by a procedure which executes an efficient prime-verification algorithm and returns true iff the given argument is prime. We are generally not interested in the complete interpretation of $isPrime$, so it can be cast as a restricted MX problem with $isPrime$ not in O. Solving such a problem using lazy grounding then has the benefit of only executing the associated function *during* search for relevant atoms $isPrime(d)$. Also for this task, we show an experimental evaluation in the next section.

6. Experiments

The IDP system has a state-of-the-art model expansion engine, as can be observed from previous Answer-Set Programming competitions (Denecker et al., 2009; Calimeri et al., 2014; Alviano et al., 2013). The lazy model expansion algorithms presented in this paper were implemented in the IDP system, by extending the existing algorithms (De Cat, Bogaerts, Devriendt, & Denecker, 2013).

The current implementation is incomplete in the sense that the cycle check for justifications has not been implemented yet. This only affects inductive definitions as non-inductive ones can be replaced by the FO formulas of their completion. As a workaround for the lack of a cycle check, build_djust, the function that constructs a direct justification, returns false for rules defining inductive predicates. As a consequence, an instance of such a rule is immediately grounded, although lazily, when a domain atom defined by the rule is assigned a value. Another consequence is that inductively defined predicates cannot be used in justifications of other rules. This affects three benchmarks of the ASP competition (de-

5. Within the ASP community, they are sometimes referred to as “show” predicates.

scribed below in Section 6.2), namely `Reachability`, `Sokoban` and `Labyrinth`. For these, the grounding might be delayed even more in a complete implementation.

The section is organized as follows. In Section 6.1, we evaluate the overhead of completely grounding a theory using the presented approach. In Section 6.2, we evaluate the effect of lazy grounding on a number of benchmarks of the ASP competition. In Section 6.3, a number of additional properties of the presented algorithms are demonstrated.

We tested three different setups: IDP with the standard ground-and-solve approach (referred to as `g&s`), IDP with lazy model expansion (`lazy`) and the award-winning ASP system Gringo-Clasp (`ASP`). We used IDP version 3.2.1-lazy, Gringo 3.0.5 and Clasp 2.1.2-st. The parameters of the lazy grounding algorithms are discussed in Section 5.1, the values used in the experiments are documented in Appendix A. The experiments for Sections 6.1 and 6.3 were run on a 64-bit Ubuntu 13.10 system with a quad-core 2.53 GHz processor and 8 GB of RAM. Experiments for Section 6.2 were run on a 64-bit Ubuntu 12.10 system with a 24-core 2.40-Ghz processor and 128 GB of RAM. A timeout of 1000 seconds and a memory limit of 3 GB was used; out-of-time is indicated by `T`, out-of-memory by `M`.[6]

6.1 Effect on Grounding Time

Lazy grounding may reduce grounding size and time but also causes overhead. For instance, we expect the (naive) incremental querying of justifications to be costly as discussed previously. The aim of this section is to quantify the overhead caused by lazy grounding. In the experiments below we compare the grounding time of the standard IDP system with that of a *naive* instance of the lazy grounding algorithm that is forced to generate the complete grounding before starting the search. This instance was obtained from the standard algorithm using some small changes: the shortcut to ground small formulas at once is turned off, disjuncts and instances of existentially quantified formulas are grounded one by one, a defined literal is enqueued for lazy grounding as soon as it appears in Δ_{g}. For comparison, we also measure the cost of the standard lazy grounding algorithm that computes partial groundings.

We devised six benchmarks to test various aspects of the novel algorithm. Each benchmark is a simple theory with at most two sentences that is simple to solve. The benchmarks are designed to measure the cost of different aspects of lazy grounding: delaying and resuming grounding, the querying needed to resume grounding, the splitting of formulas, etc. Specifically, the tested aspects are the following:

1. Overhead of delaying and resuming grounding in case of an existential quantifier with a large domain. The sentence is $\exists \overline{x} : P(\overline{x})$. Standard grounding creates a single clause with n disjuncts; naive lazy grounding grounds the formula piece by piece and introduces $n-2$ Tseitin symbols.

2. Overhead in case of an inductive definition, here $\{\forall \overline{x} : P(\overline{x}) \leftarrow P(\overline{x}) \lor Q(\overline{x})\}$. Both standard grounding and naive lazy grounding construct a ground rule for each $P(\overline{d})$ atom.

6. Benchmarks, experimental data and complete results are available at `http://dtai.cs.kuleuven.be/krr/experiments/lazygrounding/jair`.

3. Overhead in case of a universal quantification. The sentence is $\forall \overline{x} : P(\overline{x})$. While standard grounding creates n atomic formulas, naive lazy grounding splits off one instance at a time and introduces $n - 2$ Tseitin symbols.

4. Lifted Unit Propagation (LUP) (Wittocx et al., 2010, 2013) is an important preprocessing step to reduce the grounding size. Concretely, applying LUP to the rules

$$\forall \overline{x} : \neg R(\overline{x})$$
$$\forall \overline{x} : R(\overline{x}) \Rightarrow \forall \overline{y} : P(\overline{x}, \overline{y})$$

derives that the second formula follows from the first and hence does not need to be grounded at all. This theory is used to check whether LUP remains equally important in a system with lazy grounding.

5. Overhead in case of nested universal quantification. The sentence is of the form $\forall \overline{x} : R(\overline{x}) \Rightarrow \forall \overline{y} : P(\overline{x}, \overline{y})$. Standard grounding creates a formula for each instance $\overline{d}$ of $\overline{x}$ with a Tseitin for the grounding of $\forall \overline{y} : P(\overline{d}, \overline{y})$. Naive lazy grounding creates an extra Tseitin for each instance $\overline{d}$ of $\overline{x}$ and an extra set of Tseitins for the piece by piece grounding of the subformula $\forall \overline{y} : P(\overline{d}, \overline{y})$.

6. Overhead of the incremental querying in case a symbolic justification has to be validated. The sentence is $\forall \overline{x} : R(\overline{x}) \vee S(\overline{x})$, with an identical justification formula. The formula is validated by checking the falsity of the query $\exists \overline{x} : \neg R(\overline{x}) \wedge \neg S(\overline{x})$. This query is re-evaluated each time an R-atom or S-atom is falsified.

6.1.1 Results

Experiments were done for predicates P and Q with arity 3 and R and S with arity 2, and domains of size 10, 20, 30, 40 and 50. None of the predicates symbols were interpreted in the structure.

In all experiments, the overhead for the time required to solve the initial optimization problem (for the global approach) was always around 0.02 seconds, so in itself negligible. The results for the first three experiments are not shown as the differences between standard grounding and naive lazy grounding are negligible. While expected for experiment 2, for experiments 1 and 3, it shows that our actual implementation eliminates the overhead for Tseitins when quantifiers are not nested. In each of these three experiments, standard lazy grounding is able to justify the formulas without grounding them and hence fast and almost insensitive to the domain size. As shown in Figure 4, there is no difference between standard grounding and naive lazy grounding for experiment 4. In both cases, the use of LUP has a big impact on the size of the grounding and hence on the time. While experiment 1 and 3 showed that a top level quantifier does not create overhead for lazy grounding, experiment 5 shows that this does not hold anymore for nested quantifiers and that naive lazy grounding has substantial overhead when compared with standard grounding. Note that this overhead is worst case. When Tseitins can be justified, their definitions are not grounded, which explains why normal lazy grounding is faster than standard grounding and insensitive to the domain size. Experiment 6 shows that a more complex justification formula causes significant overhead for naive lazy grounding. Also here, the overhead is worst case and not

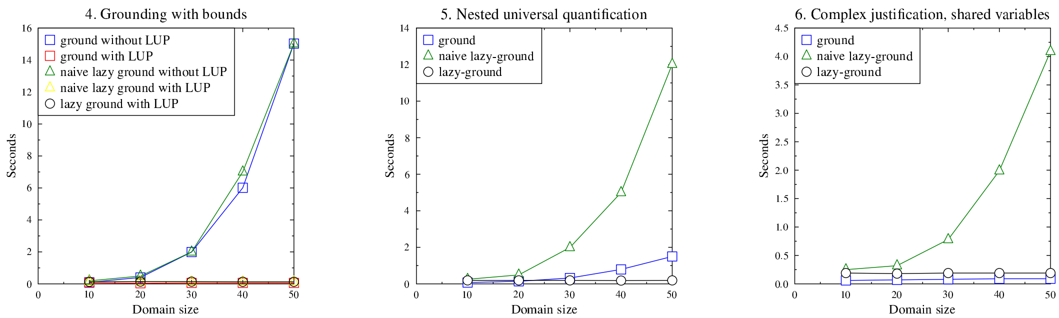

Figure 4: Time overhead of naive lazy grounding over ground-and-solve when completely grounding the input theory, for benchmarks 4, 5 and 6. The time includes grounding, solving and the time needed to find justifications. The time required by the standard lazy grounding algorithm is also shown for comparison.

visible in normal lazy grounding. Still, it is an important part of future research to reduce the overhead of the incremental querying of complex justification formulas.

6.2 ASP Competition Benchmarks

Second, we selected benchmarks from previous ASP competitions to evaluate the lazy grounding algorithm in a more realistic setting. Many benchmarks solutions of that competition are carefully fine tuned for speed and minimal grounding. Lazy grounding is usually unable to substantially reduce the grounding of such theories and, due to its overhead, is then slower than standard ground and solve. For this reason, we have sometimes selected modelings of the benchmarks that are more natural but less optimized in time and grounding size. We justify this on the ground that the aim of our work is to improve inference for declarative *modeling* (De Cat et al., 2014), where the emphasis is not on developing intricate encodings, but on modeling a problem close to its natural language specification.

We selected the following problems (see the competition websites for complete descriptions). They consist of problems that are known to be hard, in order to evaluate the effect of lazy model expansion on search, and problems that typically result in a large grounding.

- `Reachability`: Given a directed graph, determine whether a path exists between two given nodes.
- `Labyrinth`: A planning problem where an agent traverses a graph by moving between connected nodes to reach a given goal node. In addition, the graph can be manipulated to change its connectedness.
- `Packing`: Given a rectangle and a number of squares, fit all squares into the grid without overlaps.
- `Disjunctive Scheduling`: Schedule a number of actions with a given earliest start and latest end time with additional constraints on precedence and disjointness.

benchmark	# inst.	# solved			avg. time (sec.)		
		g&s	lazy	ASP	g&s	lazy	ASP
Sokoban	50	44	25	**50**	102	59	**20**
Disj. Sched.	21	5	**21**	5	130	207	**5**
Packing	50	**44**	**44**	6	173	**121**	437
Labyrinth	261	83	72	**141**	196	245	**181**
Reachability	16	2	**16**	4	110	**12**	44
Stable Marr.	106	21	**94**	5	643	402	**40**
Graph Col.	60	**34**	12	18	211	**21**	85

Table 1: The number of solved instances for the ASP benchmarks and the average time taken on the solved instances. Different solvers solve quite different sets of instances.

- `Sokoban`: A planning problem where a robot has to push a number of blocks to goal positions, constrained by a 2-D maze.
- `Graph Colouring`: Given a graph, assign colour to nodes (from a given set of colours), such that no connected nodes have the same colour.
- `Stable Marriage`: Given a set of men and women and a set of preferences, find a "stable" assignment: no swap results in a better match.

For each of these, we used all instances from the 2011 and 2013 competitions, except for the 2013 `Reachability` instances, because of the huge data files which none of the systems is designed to handle. For `Stable Marriage`, `Graph Colouring` and `Reachability`, we based our encodings on the available ASP-Core-2 encodings. For `Packing` and `Disjunctive Scheduling`, we constructed a natural $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ encoding and made a faithful translation to ASP. For the more complex benchmarks of `Labyrinth` and `Sokoban`, we used the original $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ and Gringo-Clasp's ASP specifications submitted to the 2011 competition. For the lazy model expansion, we replaced cardinality expressions by their FO encoding as for the former no justifications are derived yet; this also increases the size of the full grounding.

6.2.1 Results

The number of solved instances and average time are shown in Table 1; the average grounding size for the IDP setup is shown in Table 2.[7] For time and grounding size, unsolved instances were not taken into account. Memory overflows happened in `Reachability` (9 times for `g&s`, 9 times for `ASP`), `Disjunctive Scheduling` (6 times for `ASP`) `Labyrinth` (160 times for `g&s`, once for `ASP`), `Packing` (4 times for `g&s`, 4 times for `lazy`, 30 times for `ASP`) and `Stable Marriage` (66 times for `ASP`); all other unsolved instances were caused by a time-out.[8]

7. Grounding consists of variable instantiation interleaved with formula simplification (e.g., dropping false disjuncts, true conjuncts, replacing disjunctions with true disjuncts by true and conjunctions with false conjunctions by false, etc). These simplification steps may seriously reduce the grounding size.
8. IDP has automatic symmetry breaking, the cause of the difference between `g&s` and `ASP` for `Graph Colouring`.

benchmark	ground size (# atoms) g&s	lazy	ASP	ground time g&s(sec.)	ASP (sec.)
Sokoban	$\mathbf{2.65 \times 10^4}$	2.90×10^5	4.63×10^4	2.0	**0.3**
Disj. Sched.	5.17×10^6	2.72×10^6	$\mathbf{8.04 \times 10^5}$	129.7	**0.7**
Packing	3.86×10^7	1.69×10^7	$\mathbf{4.53 \times 10^6}$	165.6	**4.7**
Labyrinth	1.68×10^6	1.38×10^6	$\mathbf{3.55 \times 10^5}$	101.0	**2.3**
Reachability	2.87×10^7	$\mathbf{1.61 \times 10^4}$	1.35×10^6	109.7	**14.5**
Stable Marr.	2.11×10^7	1.20×10^7	$\mathbf{3.36 \times 10^6}$	642.7	**3.2**
Graph Col.	$\mathbf{1.15 \times 10^4}$	1.58×10^4	2.80×10^4	**0.1**	**0.1**

Table 2: The average grounding size for the number of *solved* instances of the ASP benchmarks, for all setups. For the `lazy` setup, the size of the final ground theory was taken. For `g&s` and `ASP`, the average grounding time is also shown.

The results show that lazy model expansion solved more instances than the other setups in four out of seven cases. In those cases, the problems also got solved significantly below the time threshold. In five out of seven cases, the (final) grounding size was smaller for lazy model expansion, orders of magnitude in one case. For `Sokoban`, `Labyrinth` and `Graph Colouring`, lazy model expansion was outperformed by ground-and-solve, indicating that the loss of information outweighed the gain of grounding less up-front. E.g., for `Sokoban`, the final `lazy` grounding size was even higher than for `g&s` (possible due to the FO encoding of cardinalities), indicating that a large part of the search space was explored. For `Stable Marriage`, the relatively small difference in grounding size between `g&s` and `lazy` leads us to believe that the different search heuristic was the main factor, not lazy grounding itself.

We also experimented with the `Airport Pickup` ASP-2011 benchmark, a fairly standard scheduling problem (transporting passengers by taxis taking into account fuel consumption) except that no upper bound on time is provided.[9] Hence any ground-and-solve approach would need to construct an infinite grounding. Applying straightforward lazy model expansion also resulted in a grounding that was too large. However, with the prototype that uses the late grounding heuristic described in Section 5.1, IDP solved one out of ten instances. For the others, grounding was not the problem, but the search took too long at each of the time intervals $1..n$ considered to get up to a sufficient n to solve the problem (even with the standard search heuristic).

The presented results show that, although often beneficial, lazy model expansion can be a considerable overhead for some hard search problems. On the other hand, while inspecting the outcome of the experiments, we observed that the class of specifications and instances solved by lazy grounding and traditional grounding only partially overlap. This suggests that it might be a good idea to integrate both approaches into a *portfolio* system. Such a system can either select heuristically whether to use ground-and-solve or lazy model expansion (based on the input) or running both in parallel, aborting either one if it uses too much memory. However, on all the problems considered, lazy model expansion could start search

9. It is possible to derive finite worst-case thresholds for the Airport Pickup problem. This is, however, not part of the original specification.

much earlier than ground-and-solve, even though it got lost more often during search. This leads us to believe that to realize the full potential of lazy grounding, more work is necessary on developing suitable heuristics (possibly user-specified ones).

6.3 Specific Experiments

In addition to the ASP competition benchmarks, some experiments were conducted using crafted benchmarks to illustrate specific properties of the lazy grounding algorithm.

The first part of Table 3 shows the results of scalability experiments. For each of the benchmarks `Packing`, `Sokoban` and `Disjunctive Scheduling`, we selected a simple problem instance and gradually extended its domain size by orders of magnitude: the size of the grid (`Packing`) or the number of time points (`Sokoban`, `Disjunctive Scheduling`). The results show that for each of the instances, lazy model expansion scales much better than the ground-and-solve strategies of IDP and Gringo-Clasp and for satisfiable as well as unsatisfiable instances. However, for `Disjunctive Scheduling` the solving time still increases significantly. The reason is that the lazy heuristics are still naive and make uninformed choices too often.

As we mentioned in the previous section, ASP competition problems typically have small groundings since running benchmarks which are too large for any system to handle does not provide a useful comparison of the systems. Hence, we also evaluated lazy model expansion on a number of crafted benchmarks where grounding is non-trivial. It is part of future work to look for more practical applications of this type. We constructed the following benchmarks:

- `Dynamic reachability`, the example described in Section 3.3.
- Lazy evaluation of `procedurally interpreted` symbols, using a simple theory over the prime numbers. As described in Section 5.2, a predicate symbol $isPrime/1$ is interpreted by a procedure that returns true if the argument is prime.
- A predicate encoding of a `function` with a huge domain.
- An experiment that simulates model generation for a theory with an unknown domain. The unknown domain is expressed by a new predicate $used/1$; quantified formulas $\forall x : \varphi$ are translated to $\forall x : (used(x) \Rightarrow \varphi)$ and $\exists x : \varphi$ to $\exists x : (used(x) \wedge \varphi)$; model generation is simulated by model expansion with a domain of size 10^6.

For each one, a faithful ASP encoding was constructed. The second part of Table 3 shows the results for these benchmarks. They show a significant improvement of lazy model expansion over ground-and-solve on all examples: in each case, both `g&s` and `ASP` went into memory overflow during grounding, while `lazy` found solutions within seconds. However, for `Disjunctive Scheduling`, it is also evident that the lazy approach would benefit from improved heuristics: increasing the domain size significantly increases the solving time, while the instances are not intrinsically harder.

6.3.1 Closer to Inherent Complexity?

During the modeling phase of an application, different encodings are typically tested out, in an attempt to improve performance or to locate bugs. While modeling our experimental

benchmark	lazy	g&s	ASP
packing-10	0.2	2.0	0.1
packing-25	0.3	2.0	0.1
packing-50	1.1	10.03	5.8
sokoban-10^3	0.31	0.3	0.1
sokoban-10^4	0.5	20.0	1.1
sokoban-10^5	2.6	T	68.0
disj-sched-sat-10^3	0.39	0.49	0.07
disj-sched-sat-10^4	13.04	16.05	17.44
disj-sched-sat-10^5	164.18	M	M
disj-sched-unsat-10^3	0.24	0 49	0.09
disj-sched-unsat-10^4	4.11	16.04	19.85
disj-sched-unsat-10^5	164.2	M	M
dynamic reachability	0.18	M	M
procedural	1.24	M	M
function	0.79	M	M
modelgeneration	0.19	M	M

Table 3: The solving time for additional crafted benchmarks, one instance each.

benchmarks, we noticed that simplifying a theory by dropping constraints often resulted in a dramatic reduction in the time lazy model expansion took to find a model. Standard model expansion, on the other hand, was much less affected by such simplifications. In our opinion, this observation, while hardly definitive evidence, is another indication that the presented algorithms are able to derive justifications for parts of a theory that can be satisfied cheaply. In that way, the approach is able to distinguish better between problems which are inherently difficult and problems which would just have a large grounding.

7. Related Work

Lazy model expansion offers a solution for the blow-up of the grounding that often occurs in the ground-and-solve model expansion methodology for FO(ID) theories. Answer Set Programming (ASP) and SAT Modulo Theories (SMT) techniques also process theories that can have a large grounding; the constraint store of Constraint Programming (CP) and Mixed Integer Programming and the clauses of SAT can be considered the equivalent of a grounded theory (they are often derived from quantified descriptions such as "$c_i < c_j$ **for all** i and j for which ...") and can also become very large. Lefèvre and Nicolas (2009) and Ge and de Moura (2009) have reported a blow-up problem in these paradigms and a multitude of techniques has been developed to address it. We distinguish four approaches.

First, concerning grounding up-front, research has been done towards *reducing the size of the grounding* itself through (**i**) *static analysis* of the input to derive bounds on variable instantiations (Wittocx et al., 2010, 2013), (**ii**) techniques to *compile* specific types of sentences into more compact ground sentences (Tamura et al., 2009; Metodi & Codish, 2012), (**iii**) detect parts that can be evaluated polynomially (Leone et al., 2006; Gebser et al., 2011; Jansen et al., 2013) and (**iv**) detect parts that are not relevant to the task at hand (e.g., in the context of query problems) as shown in the work of Leone et al. (2006). Naturally, each of these approaches can be used in conjunction with lazy grounding to further reduce the size of the grounding. In IDP, e.g., lazy grounding is already combined with (**i**) and (**iii**).

Second, the size of the grounding can be reduced by *enriching* the language. For example, ASP solvers typically support ground aggregates (interpreted second-order functions such as cardinality or sum that take sets as arguments), and CP and SMT solvers support (uninterpreted) functions. More recently, the Constraint-ASP paradigm was developed (Ostrowski & Schaub, 2012), that integrates ASP and CP by extending the ASP language with *constraint* atoms. These are interpreted as constraints in a CSP problem and can thus be handled using CP techniques. Various CASP solvers are already available, such as Clingcon (Ostrowski and Schaub), Ezcsp (Balduccini, 2011), Mingo (Liu, Janhunen, & Niemelä, 2012) and Inca (Drescher & Walsh, 2012). This technique is also integrated into IDP (De Cat et al., 2013). Inca and IDP in fact implement Lazy Clause Generation (Ohrimenko et al., 2009), an optimized form of lazy grounding for specific types of constraints. The language HEX-ASP (Eiter et al., 2005) also extends ASP, this time with *external* atoms that represent (higher-order) external function calls.

Third, *incremental approaches* are well-known from model generation, theorem proving and planning. For these tasks, the domain is typically not fixed in advance, but part of the structure being sought, such as the number of time steps in a planning problem (recall the Sokoban example from the introduction). Such an approach typically works by grounding the problem for an initial guess of (the number of elements in) the domain. Afterwards, search is applied; if no model was found, the domain is extended and more grounding is done. This is iterated until a model is found or a bound on the maximum domain size is hit (if one is known). This technique is applied, e.g., in the prover Paradox (Claessen & Sörensson, 2003) and the ASP solver IClingo (Gebser et al., 2008).

Fourth, and closest to lazy grounding itself, is a large body of research devoted to delaying the grounding of specific types of expressions until necessary (for example until they result in propagation). Propagation techniques on the first-order level that delay grounding until propagation ensues have been researched within ASP (Lefèvre & Nicolas, 2009; Dal Palù et al., 2009; Dao-Tran et al., 2012) and within CP (Ohrimenko et al., 2009). Such techniques can be used in conjunction with lazy grounding as they derive more intelligent justifications for specific types of constraints than presented here. For example, Dao-Tran et al. also presented an efficient algorithm for bottom-up propagation in a definition. Within SMT, various theory propagators work by lazily transforming their theory into SAT, such as for the theory of Bit Vectors by Bruttomesso et al. (2007). Ge and de Moura (2009) investigated quantifier handling by combining heuristic instantiation methods with research into decidable fragments of FO theories, as these can be efficiently checked for models. Within ASP, work has been done on goal-directed reasoning. Both Bonatti, Pontelli, and Son (2008) and Marple, Bansal, Min, and Gupta (2012) demonstrate approaches, in the style of SLD resolution, that apply top-down instantiation to answer queries over infinite domains. Saptawijaya and Pereira (2013) extend an abduction framework to lazily generate part of the relevant sentences. In search algorithms, justifications (or *watches*) are used to derive when a constraint will not result in propagation or is already satisfied, and hence need not be checked in the propagation phase. Nightingale et al. (2013) show how maintaining (short) justifications can significantly reduce the cost of the propagation phase.

In fact, a well-known technique already exists that combines search with lazy instantiation of quantifiers, namely *skolemization*, where existentially quantified variables are replaced by newly introduced function symbols. Universal quantifications are handled by instantiating them for those introduced function symbols. Reasoning on consistency can, e.g., be achieved by congruence closure algorithms, capable of deriving consistency without effectively assigning an interpretation to the function symbols. These techniques are used in Tableau theorem proving (Hähnle, 2001) and SMT solvers (Detlefs, Nelson, & Saxe, 2005). Formula (Jackson, Bjorner, & Schulte, 2013) interleaves creating a ground program and giving it to an SMT solver, iterating when symbolic guesses proved to be wrong. Skolemization-based techniques typically work well in case only a small number of constants needs to be introduced, but have difficulty in case the relevant domain is large. One can also see that lazy grounding (with support for function symbols) could incorporate skolemization by adapting the rules for grounding existential and universal quantification. We expect skolemization to be complementary to lazy grounding, but an in-depth investigation is part of future work.

In the field of probabilistic inference, several related techniques have been developed that also rely on lazy instantiation. First, the Problog system uses a form of static dependency analysis to ground a (probabilistic) program in the context of a given query, by constructing all possible ways to derive the query in a top-down fashion (Kimmig et al., 2011). Second, so-called *lazy inference*, applied e.g. in *LazySAT* (Singla & Domingos, 2006), exploits the fact that, for the considered inference, a (fixed) *default* assumption exists under which an expression certainly does not contribute to the probabilities. Hence, expressions for which the assumption certainly holds do not have to be considered during search. Third, *cutting plane inference* (Riedel, 2009) applies lazy inference in an interleaved setting, only constructing the part of the program for which the assumptions are not satisfied.

8. Future Work

Several aspects of the presented work need further investigation. One aspect is extending support to lazily ground more complex expressions, including aggregate expressions and (nested) function terms. Consider for example the sentence $(\sum_{x \in D \text{ and } P(x)} f(x)) > 3$, which expresses that the sum of terms $f(d)$ for which the atom $P(d)$ is true, with $d \in D$, P a predicate and f a function, should be larger than 3. One can observe that it is not necessary to ground the whole sentence up-front. For example, if f maps to the natural numbers (hence positive), the set $\{P(d_1), f(d_1) > 3\}$ is a minimal justification. Even if no easy justification can be found, we can suffice by grounding only part of the sentence and delay the remainder. For example, we can create the ground sentence $(\sum_{P(d_1)} f(d_1)) > 3 \vee T$, with T a Tseitin symbol defined as $(\sum_{P(d_1)} f(d_1)) + (\sum_{x \in D \setminus d_1 \text{ and } P(x)} f(x)) > 3$. Indeed, in any model of the sentence in which T is false, the original inequality is satisfied.

A second aspect is whether there are advantages to grounding earlier, for example to guarantee no propagation is lost, or grounding later, possibly reducing the size of the grounding even more. For example, consider the sentences $P \Rightarrow \phi$ and $\neg P \Rightarrow \psi$, with ϕ and ψ both large formulas for which no justification was found. Instead of grounding at least one of the sentences, we might add P to the list of atoms the search algorithm has to assign and only ground either of the sentences when P has been assigned a value (it might even be that unsatisfiability is detected before grounding either one).

Given that lazy grounding is useful, what about lazy *forgetting* the grounded theory? As the ground theory is extended when making the structure more precise, the ground theory could be reduced again during backtracking. By storing the justification violations that caused grounding, we can derive which grounding can be forgotten again if the violation is no longer problematic (e.g., after backtracking). For this, an algorithm needs to be developed which tracks grounding/splitting dependencies between rules given their justifications. This closely resembles techniques used in tableau theorem proving and SMT, where the theory at hand can be compacted when moving to a different part of the search space.

The approach described for lazy grounding can also be applied to answer set generation in the field of ASP. In ASP, a logic program under stable semantics can be seen as one rule set, a single definition. However, such ASP programs do not satisfy a major condition to apply lazy grounding. Indeed such programs are typically non-total, due to the presence of constraints and rules of the form $p \leftarrow \mathit{not}\ np, np \leftarrow \mathit{not}\ p$ or other *choice rules* which result in multiple stable models. However, as described by Denecker et al. (2012), most practical ASP programs can be partitioned in a set of choice rules, a set of *total* definitions and a set of constraints (the so-called Generate-Define-Test partition). Any ASP program that can be GDT-partitioned, can be translated straightforwardly into an equivalent FO(ID) theory that only contains total definitions. This suggests a way to apply lazy grounding to such ASP programs.

9. Conclusion

Solvers used in the domains of SAT, SMT and ASP are often confronted with problems that are too large to ground. Lazy model expansion, the technique described in this paper, interleaves grounding and search in order to avoid the grounding bottleneck. The technique builds upon the concept of a justification, a deterministic recipe to extend an interpretation such that it satisfies certain constraints. A theoretical framework has been developed for lazy model expansion for the language FO(ID) and algorithms have been presented to derive and maintain such justifications and to interleave grounding with state-of-the-art CDCL search algorithms. The framework aims at bounded model expansion, in which all domains are finite, but is also an initial step towards handling infinite domains efficiently. Experimental evaluation has been provided, using an implementation in the IDP system, in which lazy model expansion was compared with a state-of-the-art ground-and-solve approach. The experiments showed considerable improvement over ground-and-solve in existing benchmarks as well as in new applications. The main disadvantage is the less-informed search algorithm, caused by the delay in propagation and the introduction of additional symbols. A possible solution is to develop new heuristics or portfolio approaches that combine the strengths of both methods. Finally, we have indicated a way how the proposed methods can be applied beyond FO(ID), to ASP solvers in general.

Acknowledgements

During this research, Broes De Cat was funded by the Agency for Innovation by Science and Technology in Flanders (IWT). This research was also supported by FWO-Vlaanderen and by the project GOA 13/010, Research Fund KULeuven. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

Appendix A. More Details about the Algorithms

In this appendix, we mention parameter values as well as some optimizations that further reduce the grounding overhead and/or improve the search. For each optimization, we indicate what is currently implemented (and part of the experimental results) and what is part of future work.

A.1 Parameter Values

In 5.1, a number of parameters were introduced to control the behavior of lazy model expansion. Here, we provide details on the values used in the experimental evaluation. These values were set manually, based on experience and a limited number of observations (e.g., the extension threshold works similar to the conflict threshold of the SAT solver). It is part of future work to study the impact of different values.

- For an existential quantification, 10 instantiations are grounded at a time; for a disjunction, 3 disjuncts are grounded at a time. This turned out to give the best balance between introducing too many Tseitin atoms and grounding too much.
- The initial truth value is **t** with probability 0.2 and **f** otherwise.
- The initial threshold for randomized restarts is 100 extensions of the ground theory. It is doubled after each restart.
- A formula is considered small if its estimated grounding size is below 10^4 atoms.

A.2 Extension to $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$

So far, we have described a lazy model expansion algorithm for function-free $\mathrm{FO}(ID)$. However, $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, the knowledge-base language of the IDP system, supports a much richer input language. Besides types —which we use to initialize the domains— it also supports (partial) functions, aggregates and arithmetic. Our current implementation ignores the latter extensions through a straightforward adaptation of build_djust (Algorithm 4): the case for literals is extended to return *false* when the literal is not part of the function-free $\mathrm{FO}(ID)$ language. For example, given a rule $h \leftarrow \forall x : P(x) \lor Q(f(x))$, $P(x)$ can be used in a justification but $Q(f(x))$ cannot. For functions, there is also the option to replace them by graph predicates during the preprocessing step. As for the experiments of Section 6.2, functions, if any, are given in the input structure and hence play no role.

It is part of future work to extend lazy grounding for these extensions, especially for functions. Techniques developed in SMT and in Constraint Programming to handle (ground) atoms containing function symbols are useful to reduce the size of the grounding and improve search. In previous work, these techniques have been integrated in the IDP system (De Cat et al., 2013) and it is certainly worthwhile to fully integrate them with lazy grounding.

A.3 Cheap Propagation Checks.

In lazy_mx, it is checked for each assigned literal whether it is defined in Δ_{d} and whether it violates any justifications. To implement this cheaply, our implementation maintains a mapping for literals in Δ_{g}. It states whether the literal is defined in Δ_{d} and also lists the justifications in which its negation occurs. This mapping is extended whenever a new literal is added to Δ_{g} and maintained whenever justifications change. The performance of the search loop is unaffected as long as literals are assigned for which the mapping is empty.

A.4 Stopping Early

In Algorithm 2, we took the standard stopping criterion used in most search algorithms (Line 14): to stop in a conflict-free state where $\mathcal{I}$ is two-valued on all symbols of $T_{\mathrm{g}} \cup \Delta_{\mathrm{g}}$. In principle, we may stop earlier, with a partial structure $\mathcal{I}$ that admits a total justification for $P_{\mathcal{T}}$. Indeed, Corollary 3.6 tells us that such an $\mathcal{I}$ can be expanded to a model. This has a considerable impact on grounding size. Indeed, assigning a truth value to an atom A defined in Δ_{d} that is irrelevant (in effect, does not appear in the justification) will trigger grounding of A's definition, which in turn might introduce new literals defined in Δ_{d}, causing a cascade of unnecessary groundings and assignments. Our solver algorithm does not maintain a justification of Δ_{g}, so it cannot know exactly when a justification exists. Instead, the implemented algorithm only chooses literals that are watched by some formula/rule. It stops with a partial structure in which unwatched literals may not be assigned. It can be shown that this suffices to guarantee that $\mathcal{I}$ admits a justification. Hence it is safe to stop search.

A.5 Approximate Justifications

In some cases, build_djust cannot find a valid justification for a large formula because a few literals are already false in $\mathcal{I}$. For example for a formula $\forall x \in D : P(x)$, build_djust returns *false* if at least one atom of P is false. Instead, we have adapted build_djust with a heuristic check on the number of expected violations. If it is small enough, the justification is still returned. Naturally, we are then required to check whether there are any real violations, by querying the justification formula over $\mathcal{I}$, and apply lazy_ground to them.

References

Alviano, M., Calimeri, F., Charwat, G., Dao-Tran, M., Dodaro, C., Ianni, G., Krennwallner, T., Kronegger, M., Oetsch, J., Pfandler, A., Pührer, J., Redl, C., Ricca, F., Schneider, P., Schwengerer, M., Spendier, L. K., Wallner, J. P., & Xiao, G. (2013). The fourth Answer Set Programming competition: Preliminary report. In Cabalar, P., & Son, T. C. (Eds.), *LPNMR*, Vol. 8148 of *LNCS*, pp. 42–53. Springer.

Apt, K. R. (2003). *Principles of Constraint Programming*. Cambridge University Press.

Balduccini, M. (2011). Industrial-size scheduling with ASP+CP. In Delgrande, J. P., & Faber, W. (Eds.), *LPNMR*, Vol. 6645 of *LNCS*, pp. 284–296. Springer.

Baral, C. (2003). *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA.

Bonatti, P. A., Pontelli, E., & Son, T. C. (2008). Credulous resolution for answer set programming. In Fox, D., & Gomes, C. P. (Eds.), *AAAI*, pp. 418–423. AAAI Press.

Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Hanna, Z., Nadel, A., Palti, A., & Sebastiani, R. (2007). A lazy and layered SMT(BV) solver for hard industrial verification problems. In Damm, W., & Hermanns, H. (Eds.), *CAV*, Vol. 4590 of *LNCS*, pp. 547–560. Springer.

Calimeri, F., Ianni, G., & Ricca, F. (2014). The third open answer set programming competition. *TPLP*, *14*(1), 117–135.

Chen, W., & Warren, D. S. (1996). Tabled evaluation with delaying for general logic programs. *J. ACM*, *43*(1), 20–74.

Claessen, K., & Sörensson, N. (2003). New techniques that improve MACE-style model finding. In *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*.

Dal Palù, A., Dovier, A., Pontelli, E., & Rossi, G. (2009). Answer set programming with constraints using lazy grounding. In Hill, P. M., & Warren, D. S. (Eds.), *ICLP*, Vol. 5649 of *LNCS*, pp. 115–129. Springer.

Dao-Tran, M., Eiter, T., Fink, M., Weidinger, G., & Weinzierl, A. (2012). Omiga : An open minded grounding on-the-fly answer set solver. In del Cerro, L. F., Herzig, A., & Mengin, J. (Eds.), *JELIA*, Vol. 7519 of *LNCS*, pp. 480–483. Springer.

De Cat, B. (2014). *Separating Knowledge from Computation: An FO(.) Knowledge Base System and its Model Expansion Inference*. Ph.D. thesis, KU Leuven, Leuven, Belgium.

De Cat, B., Bogaerts, B., Bruynooghe, M., & Denecker, M. (2014). Predicate logic as a modelling language: The IDP system. *CoRR*, *abs/1401.6312*.

De Cat, B., Bogaerts, B., Devriendt, J., & Denecker, M. (2013). Model expansion in the presence of function symbols using constraint programming. In *ICTAI*, pp. 1068–1075. IEEE.

De Cat, B., Denecker, M., & Stuckey, P. J. (2012). Lazy model expansion by incremental grounding. In Dovier, A., & Costa, V. S. (Eds.), *ICLP (Technical Communications)*, Vol. 17 of *LIPIcs*, pp. 201–211. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Delgrande, J. P., & Faber, W. (Eds.). (2011). *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, Vol. 6645 of *LNCS*. Springer.

Denecker, M. (1998). The well-founded semantics is the principle of inductive definition. In Dix, J., del Cerro, L. F., & Furbach, U. (Eds.), *JELIA*, Vol. 1489 of *LNCS*, pp. 1–16. Springer.

Denecker, M. (2000). Extending classical logic with inductive definitions. In Lloyd, J. W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Palamidessi, C., Pereira, L. M., Sagiv, Y., & Stuckey, P. J. (Eds.), *CL*, Vol. 1861 of *LNCS*, pp. 703–717. Springer.

Denecker, M., Bruynooghe, M., & Marek, V. W. (2001). Logic programming revisited: Logic programs as inductive definitions. *ACM Trans. Comput. Log.*, *2*(4), 623–654.

Denecker, M., & De Schreye, D. (1992). Justification semantics: A unifying framework for the semantics of logic programs. Tech. rep. 157, Department of Computer Science, K.U.Leuven.

Denecker, M., & De Schreye, D. (1993). Justification semantics: A unifying framework for the semantics of logic programs. In Pereira, L. M., & Nerode, A. (Eds.), *LPNMR*, pp. 365–379. MIT Press.

Denecker, M., Lierler, Y., Truszczynski, M., & Vennekens, J. (2012). A Tarskian informal semantics for answer set programming. In Dovier, A., & Costa, V. S. (Eds.), *ICLP (Technical Communications)*, Vol. 17 of *LIPIcs*, pp. 277–289. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Denecker, M., & Ternovska, E. (2008). A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log.*, *9*(2), 14:1–14:52.

Denecker, M., & Vennekens, J. (2014). The well-founded semantics is the principle of inductive definition, revisited. In Baral, C., De Giacomo, G., & Eiter, T. (Eds.), *KR*, pp. 22–31. AAAI Press.

Denecker, M., Vennekens, J., Bond, S., Gebser, M., & Truszczyński, M. (2009). The second answer set programming competition. In Erdem, E., Lin, F., & Schaub, T. (Eds.), *LPNMR*, Vol. 5753 of *LNCS*, pp. 637–654. Springer.

Detlefs, D., Nelson, G., & Saxe, J. B. (2005). Simplify: A theorem prover for program checking. *J. ACM*, *52*(3), 365–473.

Dovier, A., & Costa, V. S. (Eds.). (2012). *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary. Proceedings*, Vol. 17 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Drescher, C., & Walsh, T. (2012). Answer set solving with lazy nogood generation. In Dovier, A., & Costa, V. S. (Eds.), *ICLP (Technical Communications)*, Vol. 17 of *LIPIcs*, pp. 188–200. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Eiter, T., Ianni, G., Schindlauer, R., & Tompits, H. (2005). A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In Kaelbling, L. P., & Saffiotti, A. (Eds.), *IJCAI*, pp. 90–96. Professional Book Center.

Enderton, H. B. (2001). *A Mathematical Introduction To Logic* (Second edition). Academic Press.

Erdem, E., Lin, F., & Schaub, T. (Eds.). (2009). *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, Vol. 5753 of *LNCS*. Springer.

Ge, Y., & de Moura, L. M. (2009). Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Bouajjani, A., & Maler, O. (Eds.), *CAV*, Vol. 5643 of *LNCS*, pp. 306–320. Springer.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., & Thiele, S. (2008). Engineering an incremental ASP solver. In García de la Banda, M., & Pontelli, E. (Eds.), *ICLP*, Vol. 5366 of *LNCS*, pp. 190–205. Springer.

Gebser, M., Kaminski, R., König, A., & Schaub, T. (2011). Advances in Gringo series 3. In Delgrande, J. P., & Faber, W. (Eds.), *LPNMR*, Vol. 6645 of *LNCS*, pp. 345–351. Springer.

Gebser, M., Schaub, T., & Thiele, S. (2007). GrinGo : A new grounder for Answer Set Programming. In Baral, C., Brewka, G., & Schlipf, J. S. (Eds.), *LPNMR*, Vol. 4483 of *LNCS*, pp. 266–271. Springer.

Hähnle, R. (2001). Tableaux and related methods. In Robinson, J. A., & Voronkov, A. (Eds.), *Handbook of Automated Reasoning*, pp. 100–178. Elsevier and MIT Press.

Jackson, E. K., Bjorner, N., & Schulte, W. (2013). Open-world logic programs: A new foundation for formal specifications. Tech. rep. MSR-TR-2013-55, Microsoft Research.

Jansen, J., Jorissen, A., & Janssens, G. (2013). Compiling input$*$ FO$(\cdot)$ inductive definitions into tabled Prolog rules for IDP^3. *TPLP*, *13*(4-5), 691–704.

Karp, R. (1972). Reducibility among combinatorial problems. In Miller, R., & Thatcher, J. (Eds.), *Complexity of Computer Computations*, pp. 85–103. Plenum Press.

Kimmig, A., Demoen, B., De Raedt, L., Santos Costa, V., & Rocha, R. (2011). On the implementation of the probabilistic logic programming language ProbLog. *TPLP*, *11*(2-3), 235–262.

Lefèvre, C., & Nicolas, P. (2009). The first version of a new ASP solver: ASPeRiX. In Erdem, E., Lin, F., & Schaub, T. (Eds.), *LPNMR*, Vol. 5753 of *LNCS*, pp. 522–527. Springer.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., & Scarcello, F. (2006). The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, *7*(3), 499–562.

Liu, G., Janhunen, T., & Niemelä, I. (2012). Answer Set Programming via Mixed Integer Programming. In Brewka, G., Eiter, T., & McIlraith, S. A. (Eds.), *KR*, pp. 32–42. AAAI Press.

Marek, V. W., & Truszczyński, M. (1999). Stable models and an alternative logic programming paradigm. In Apt, K. R., Marek, V. W., Truszczyński, M., & Warren, D. S. (Eds.), *The Logic Programming Paradigm: A 25-Year Perspective*, pp. 375–398. Springer-Verlag.

Mariën, M. (2009). *Model Generation for ID-Logic*. Ph.D. thesis, Department of Computer Science, KU Leuven, Belgium.

Mariën, M., Gilis, D., & Denecker, M. (2004). On the relation between ID-Logic and answer set programming. In Alferes, J. J., & Leite, J. A. (Eds.), *JELIA*, Vol. 3229 of *LNCS*, pp. 108–120. Springer.

Mariën, M., Wittocx, J., Denecker, M., & Bruynooghe, M. (2008). SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In Kleine Büning, H., & Zhao, X. (Eds.), *SAT*, Vol. 4996 of *LNCS*, pp. 211–224. Springer.

Marple, K., Bansal, A., Min, R., & Gupta, G. (2012). Goal-directed execution of answer set programs. In Schreye, D. D., Janssens, G., & King, A. (Eds.), *PPDP*, pp. 35–44. ACM.

Marques Silva, J. P., Lynce, I., & Malik, S. (2009). Conflict-driven clause learning SAT solvers. In Biere, A., Heule, M., van Maaren, H., & Walsh, T. (Eds.), *Handbook of Satisfiability*, Vol. 185 of *Frontiers in Artificial Intelligence and Applications*, pp. 131–153. IOS Press.

Metodi, A., & Codish, M. (2012). Compiling finite domain constraints to SAT with BEE. *TPLP*, *12*(4-5), 465–483.

Mitchell, D. G., & Ternovska, E. (2005). A framework for representing and solving NP search problems. In Veloso, M. M., & Kambhampati, S. (Eds.), *AAAI*, pp. 430–435. AAAI Press / The MIT Press.

Mitchell, D. G., Ternovska, E., Hach, F., & Mohebali, R. (2006). Model expansion as a framework for modelling and solving search problems. Tech. rep. TR 2006-24, Simon Fraser University, Canada.

Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., & Tack, G. (2007). Minizinc: Towards a standard CP modelling language. In Bessiere, C. (Ed.), *CP'07*, Vol. 4741 of *LNCS*, pp. 529–543. Springer.

Nightingale, P., Gent, I. P., Jefferson, C., & Miguel, I. (2013). Short and long supports for constraint propagation. *J. Artif. Intell. Res. (JAIR)*, *46*, 1–45.

Ohrimenko, O., Stuckey, P. J., & Codish, M. (2009). Propagation via lazy clause generation. *Constraints*, *14*(3), 357–391.

Ostrowski, M., & Schaub, T. (2012). ASP modulo CSP: The clingcon system. *TPLP*, *12*(4-5), 485–503.

Riedel, S. (2009). Cutting plane MAP inference for Markov logic. In *International Workshop on Statistical Relational Learning (SRL-2009)*.

Saptawijaya, A., & Pereira, L. M. (2013). Towards practical tabled abduction in logic programs. In Correia, L., Reis, L. P., & Cascalho, J. (Eds.), *EPIA*, Vol. 8154 of *LNCS*, pp. 223–234. Springer.

Singla, P., & Domingos, P. (2006). Memory-efficient inference in relational domains. In Gil, Y., & Mooney, R. J. (Eds.), *AAAI*, pp. 488–493. AAAI Press.

Son, T. C., Pontelli, E., & Le, T. (2014). Two applications of the ASP-Prolog system: Decomposable programs and multi-context systems. In Flatt, M., & Guo, H.-F. (Eds.), *PADL*, Vol. 8324 of *Lecture Notes in Computer Science*, pp. 87–103. Springer.

Tamura, N., Taga, A., Kitagawa, S., & Banbara, M. (2009). Compiling finite linear CSP into SAT. *Constraints*, *14*(2), 254–272.

Torlak, E., Chang, F. S.-H., & Jackson, D. (2008). Finding minimal unsatisfiable cores of declarative specifications. In Cuéllar, J., Maibaum, T. S. E., & Sere, K. (Eds.), *FM*, Vol. 5014 of *LNCS*, pp. 326–341. Springer.

Tseitin, G. S. (1968). On the complexity of derivation in propositional calculus. In Slisenko, A. O. (Ed.), *Studies in Constructive Mathematics and Mathematical Logic II*, pp. 115–125. Consultants Bureau, N.Y.

Van Gelder, A. (1993). The alternating fixpoint of logic programs with negation. *J. Comput. Syst. Sci.*, *47*(1), 185–221.

Vennekens, J., Mariën, M., Wittocx, J., & Denecker, M. (2007). Predicate introduction for logics with a fixpoint semantics. Part I: Logic programming. *Fundamenta Informaticae*, *79*(1-2), 187–208.

Wittocx, J., Denecker, M., & Bruynooghe, M. (2013). Constraint propagation for first-order logic and inductive definitions. *ACM Trans. Comput. Logic*, *14*(3), 17:1–17:45.

Wittocx, J., Mariën, M., & Denecker, M. (2008). The IDP system: A model expansion system for an extension of classical logic. In Denecker, M. (Ed.), *LaSh*, pp. 153–165. ACCO.

Wittocx, J., Mariën, M., & Denecker, M. (2010). Grounding FO and FO(ID) with bounds. *J. Artif. Intell. Res. (JAIR)*, *38*, 223–269.