

Model Expansion in the Presence of Function Symbols Using Constraint Programming

Broes De Cat

Bart Bogaerts

Jo Devriendt

Marc Denecker

Report CW 644, July 2013



KU Leuven

Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Model Expansion in the Presence of Function Symbols Using Constraint Programming

Broes De Cat

Bart Bogaerts

Jo Devriendt

Marc Denecker

Report CW 644, July 2013

Department of Computer Science, KU Leuven

Abstract

The traditional approach to model expansion (generating models of a logic theory extending a partial structure) is to reduce the theory to a propositional language and apply a search algorithm to the resulting theory. Function symbols are typically replaced by predicate symbols representing the graph of the function, an operation that blows up the reduced theory.

In this paper, we present an improved approach to handle function symbols in a ground-and-solve methodology, building on ideas from Constraint Programming. We do so in the context of $\text{FO}(\cdot)^{IDP}$, the knowledge representation language that extends First-Order logic with, among others, inductive definitions, arithmetic and aggregates. A model expansion algorithm is developed, consisting of (i) a grounding algorithm for $\text{FO}(\cdot)^{IDP}$ that is parametrized by the function symbols the are allowed to occur in the reduced theory, and (ii) a search algorithm for unrestricted, ground $\text{FO}(\cdot)^{IDP}$. The ideas are implemented within the IDP knowledge-base system and experimental evaluation shows that both more compact groundings and improved search performance are obtained.

Keywords : Extended First-Order Logic, Declarative Modeling, Constraint Programming.

CR Subject Classification : F.4.1, I.2.4.

Model Expansion in the Presence of Function Symbols Using Constraint Programming

Broes De Cat, Bart Bogaerts, Jo Devriendt and Marc Denecker
 Department of Computer Science, KU Leuven, Belgium
 Email: firstname.lastname@cs.kuleuven.be

Abstract—The traditional approach to model expansion (generating models of a logic theory extending a partial structure) is to reduce the theory to a propositional language and apply a search algorithm to the resulting theory. Function symbols are typically replaced by predicate symbols representing the graph of the function, an operation that blows up the reduced theory.

In this paper, we present an improved approach to handle function symbols in a ground-and-solve methodology, building on ideas from Constraint Programming. We do so in the context of $\text{FO}(\cdot)^{\text{IDP}}$, the knowledge representation language that extends First-Order Logic (FO) with, among others, inductive definitions, arithmetic and aggregates. A model expansion algorithm is developed, consisting of (i) a grounding algorithm for $\text{FO}(\cdot)^{\text{IDP}}$ that is parametrized by the function symbols the are allowed to occur in the reduced theory, and (ii) a search algorithm for unrestricted, ground $\text{FO}(\cdot)^{\text{IDP}}$. The ideas are implemented within the IDP knowledge-base system and experimental evaluation shows that both more compact groundings and improved search performance are obtained.

I. INTRODUCTION

Model generation is a widely used problem solving paradigm. A problem is specified as a theory in a declarative logic in such a way that models of the theory represent solutions to the problem. A closely related paradigm is bounded Model Expansion (MX). Here, a partial input structure over a finite and known domain is extended into a complete structure that is a model of a given theory. These paradigms are studied and applied in the fields of Constraint Programming (CP) [1], Answer Set Programming (ASP) [26] and Knowledge Representation (KR) [3].

A state-of-the-art approach is to reduce the input theory, formulated in an expressive logic, in a model-equivalence preserving way to a theory in a fragment of the language supported by some search algorithm. Afterwards, this algorithm searches for models of the theory. For example, model generation/expansion for the language $\text{FO}(\cdot)$ [9] is performed by reducing theories to a ground fragment of $\text{FO}(\cdot)$ for which a search algorithm is available. The term *grounding* refers to both the reduction process and to its outcome; the 2-step approach is called *ground-and-solve*.

A first generation of MX systems used search algorithms for (pseudo)-propositional languages, such as Clausal Normal Form (SAT solvers) and ground ASP (ASP solvers). An important bottleneck of such systems is the blowup caused by grounding the input theory, as the size of the theory increases

rapidly with the size of the domain and the nesting depth of quantified variables. One apparent approach to reduce the nesting depth of quantified variables is to replace predicate symbols with function symbols wherever possible, as follows.

Example I.1. Consider the well-known 2-D *packing*-problem for squares: given a set of squares with known size and a rectangular area of known size, position all squares in a non-overlapping fashion within the area (if possible). One of the constraints, that two squares should not overlap horizontally, can be expressed as follows, using $\text{pos}_x(id, x)$ to express that the left side square id is at x and $\text{sz}(id)$ for the size of id :

$$\forall id_1 id_2 x_1 x_2 : id_1 \neq id_2 \wedge \text{pos}_x(id_1, x_1) \wedge \text{pos}_x(id_2, x_2) \\ \Rightarrow (x_1 + \text{sz}(id_1) \leq x_2 \vee x_2 + \text{sz}(id_2) \leq x_1).$$

In fact, pos_x represents a function mapping squares to x -coordinates, so it can be rewritten using a function $f_x(id) : x$:

$$\forall id_1 id_2 : id_1 \neq id_2 \Rightarrow (f_x(id_1) + \text{sz}(id_1) \leq f_x(id_2) \\ \vee f_x(id_2) + \text{sz}(id_2) \leq f_x(id_1)).$$

Next to being a more natural way to express the constraint, the rewriting halves the quantifier depth. However, if the target solver only takes propositional input, the function symbols are eliminated again during the reduction phase, replacing function symbols by predicate symbols and adding additional quantifiers. In fact, in the example, it comes down to transforming the latter sentence into the former one.

Recently, research is being done in ASP to incorporate techniques from CP, giving rise to the field of ASP modulo CSP (CASP) [27]. In CASP, the ASP language is extended with *constraint atoms*, atoms that stand for the constraints of a CSP problem [19][14], and can, for example, contain function symbols. Second, search algorithms have been developed that allow ground constraint atoms (instead of only propositional atoms) in the input. This gives rise to more compact groundings that often also yield better propagation. Among those next generation systems are the systems Clingcon[27], EZ(CSP)[2], Mingo[20] and Inca[12].

In this paper, we work in the context of the language $\text{FO}(\cdot)^{\text{IDP}}$, the language of the knowledge-base system (KBS) IDP [18]. $\text{FO}(\cdot)^{\text{IDP}}$ extends FO with, among others, inductive definitions, aggregates and arithmetic. We show that for $\text{FO}(\cdot)^{\text{IDP}}$, allowing the grounding to contain function terms in fact produces a general form of such “constraint atoms”, without extending the language. In the above example, $f_x(id_1) + \text{sz}(id_1) \leq f_x(id_2)$ is such an atom, for which efficient propagation techniques exist in the field of CP. We present a model expansion algorithm for $\text{FO}(\cdot)^{\text{IDP}}$ that exploits

Broes De Cat is funded by the Institute for Innovation through Science and Technology Flanders (IWT).

this idea. It of (i) an algorithm to ground $\text{FO}(\cdot)^{\text{IDP}}$ theories without eliminating all function symbols from the grounding and (ii) a search algorithm for general, ground $\text{FO}(\cdot)^{\text{IDP}}$. As different search algorithms often support different sets of function symbols, the grounding algorithm is *parametrized* by the set of functions allowed to occur in the grounding. The search algorithm, extends the search algorithm of the state-of-the-art solver MINISAT(ID) [21] using the technique of lazy clause generation[30], an approach to support finite-domain constraints in a SAT-solver by encoding propagation as clauses (for details, see Section IV). The algorithms are implemented within IDP (i) and as the solver CONSTRAINT-ID (ii).

We take terminology from the logic-based point of view to model generation. Below, we provide a short overview of coinciding notions from CP and ASP¹. A *theory* \mathcal{T} can be seen as a set of constraints (CP) or a logic program (ASP). Symbols are by default non-defined / uninterpreted; *constants* (0-ary functions symbols) coincide with variables in the Constraint Programming sense (CP-variables) and *n*-ary ($n > 0$) *function symbols* can be seen as *n*-dimensional arrays of CP-variables. A (partial) *interpretation* coincides with a (partial) assignment to CP-variables; a *model* of \mathcal{T} is a total interpretation satisfying \mathcal{T} , i.e., a solution (CP) or answer set (ASP). A *domain* is a set of domain elements, e.g., the set of values a CP-variable might take. Following CP-terminology, the domain of a function symbol refers to the set of values it can map to. A *variable* is a placeholder for instantiation with domain elements.

The paper is organized as follows. In Section II, the language $\text{FO}(\cdot)^{\text{IDP}}$ is introduced. Next, the algorithms for grounding and search are presented in Section III, respectively Section IV. Experimental evaluation is presented in Section V; related work and concluding remarks in Section VI.

II. PRELIMINARIES

We assume familiarity with FO. The notation $\text{FO}(\cdot)$ [9] denotes the family of extensions of FO with new language constructs. The language we consider in this paper is the language $\text{FO}(\cdot)^{\text{IDP}}$. It is a many-sorted extension of FO with aggregate functions, arithmetic and inductive definitions. We know give an overview of the language.

A vocabulary Σ consists of a set Σ_t of types denoted τ and a set Σ_s of typed predicate symbols denoted P, Q, R and function symbols denoted f, g, h . For each type τ , Σ includes a unary predicate symbol $T(\tau)$ representing all elements in τ .

Variables x, y , terms t , atoms A , literals L , domain elements d , and FO-formulas φ are defined as usual. A *domain atom* (*domain term*) is an atom (term) consisting of a predicate (function) symbol applied to a tuple of domain elements. We use c to denote domain terms and e to denote domain elements or domain terms. The set of symbols of a theory \mathcal{T} is denoted $\text{voc}(\mathcal{T})$. Given two tuples \bar{x} and \bar{x}' of terms of equal length n , $\bar{x} = \bar{x}'$ denotes the conjunction $x_1 = x'_1 \wedge \dots \wedge x_n = x'_n$. A term t containing occurrences of a term t' is denoted as $t[t']$; the replacement of t' in t by t'' is denoted as $t[t'/t'']$ (similarly for formulas).

An interpretation for a type τ is a set of domain elements D_τ . A (partial) interpretation for a predicate symbol $P(\bar{\tau})$

consists of two disjoint subsets of $D_{\tau_1} \times \dots \times D_{\tau_n}$, denoted as P_{ct} and P_{cf} . An (partial) interpretation for a function symbol $f(\bar{\tau})$: τ' is a function mapping elements of $D_{\tau_1} \times \dots \times D_{\tau_n}$ to a non-empty subset of τ' . A (partial) Σ -interpretation \mathcal{I} is then an interpretation for all symbols in Σ ; we use $s^{\mathcal{I}}$ to refer to the interpretation of a symbol s in \mathcal{I} . An atom $P(\bar{d})$ is true in \mathcal{I} if $P(\bar{d}) \in P_{ct}^{\mathcal{I}}$, false if $P(\bar{d}) \in P_{cf}^{\mathcal{I}}$ and unknown otherwise. An atom $f(\bar{d}) = d'$ is true in \mathcal{I} if $\{d'\} = f^{\mathcal{I}}(\bar{d})$, unknown if $\{d'\} \subsetneq f^{\mathcal{I}}(\bar{d})$ and false otherwise. An interpretation for a predicate symbol P is two-valued if $P_{ct} = P_{cf}$; an interpretation for a function symbol is two-valued if all images are singletons. An interpretation is two-valued if the interpretation of all its symbols are two-valued. For a two-valued interpretation \mathcal{I} , $\varphi^{\mathcal{I}}$ ($t^{\mathcal{I}}$) denotes the value of a formula φ (a term t) under \mathcal{I} as usual. A well-typed expression is one in which the type of each argument matches with the type of its argument position. Badly typed atoms are false. We say a term t (formula φ) is *ground-evaluatable* in \mathcal{I} if all symbols in t are two-valued. In this paper, we only consider interpretations where all types are finitely interpreted and with a total order on all domain elements; in that case, define min_τ as the function mapping to the smallest element of τ and pred_τ as the (partial) predecessor function over τ .

We extend the notion of term to include aggregate terms. A set expression is of the form $\{\bar{x} : \varphi : t\}$, $\{\varphi : t\}$ (if there are no local variables) or a union of set expressions $\{\bar{x}_1 : \varphi_1 : t_1\} \cup \{\bar{x}_2 : \varphi_2 : t_2\}$ (denoted shortly as $\{\bar{x}_1 : \varphi_1 : t_1, \bar{x}_2 : \varphi_2 : t_2\}$). Given an interpretation I and an assignment \bar{d} to the free variables \bar{y} of the set expression, the interpretation $\{\bar{x} : \varphi[\bar{y}/\bar{d}] : t[\bar{y}/\bar{d}]\}^I$ is the multiset $\{t[\bar{x}/\bar{d}', \bar{y}/\bar{d}]^{\mathcal{I}} \mid \varphi[\bar{x}/\bar{d}', \bar{y}/\bar{d}]^{\mathcal{I}} = \mathbf{t}\}$. Thus, in the context of a given assignment for the variables \bar{y} , the expression denotes the multiset of tuples t for which φ holds. Aggregate terms are of the form $\text{agg}(S)$, with agg an aggregate function (cardinality, sum, product, minimum or maximum) and S a set expression. The cardinality function then maps a set interpretation to the number of elements in the set. The aggregate functions sum, product, minimum and maximum map a set to respectively the sum, product, minimum and maximum of the elements in the set, or to 0, respectively 1, $+\infty$ and $-\infty$ if the set is empty. Aggregate terms can occur nested in other aggregates; in this paper however, aggregate terms occurring in a definition cannot contain any symbols defined in that definition.²

Definitions Δ are sets of rules of the form $\forall \bar{x} : P(\bar{t}) \leftarrow \varphi$, where $P(\bar{t})$ is called the *head* and φ the body of the rule. Predicates in the head of rules of Δ are called *defined predicates*; all other symbols in Δ are called *parameters* or *open symbols* of Δ . Intuitively, for each value of the parameters, Δ defines the defined predicates in a unique way. The satisfaction relation of FO is extended to definitions. We say that I satisfies Δ ($I \models \Delta$) if I is the parametrised well-founded model of Δ [29]. The well-founded semantics is used here because it correctly formalises the most common forms of informal inductive definitions (monotone inductive definitions and definitions over a well-founded order). Definitional implication \leftarrow should not be confused with the material implications \Leftarrow and \Rightarrow . Intuitively, when the condition of a material implication is false, its head is arbitrary (true or false), while if the condition

¹A formal comparison of $\text{FO}(\cdot)$ and ASP is presented in [8]

²In fact, the results in this paper are also correct for such aggregate terms if they do not occur nested in other aggregate terms.

of a definitional rule is false, its head cannot be derived and is false (unless another rule derives it). This intuition coincides exactly with inductive definitions as in mathematical texts.

The completion of Δ for a symbol P , defined in Δ by the rules $\forall \bar{x}_i : P(\bar{t}_i) \leftarrow \varphi_i$ with $i \in [1, n]$, is the set consisting of the sentence $\forall \bar{x}_i : \varphi_i \Rightarrow P(\bar{t}_i)$ for each $i \in [1, n]$ and the sentence $\forall \bar{x} : P(\bar{x}) \Rightarrow \bigvee_{i \in [1, n]} (\bar{x} = \bar{t}_i \wedge \varphi_i)$. This set is denoted as $comp_{P, \Delta}$, the union of all these sets for Δ as $comp_{\Delta}$. It is well-known that if $I \models \Delta$ then $I \models comp_{\Delta}$ but not vice-versa (e.g., the inductive definition expressing transitive closure is stronger than its completion).

For a vocabulary Σ and a structure \mathcal{I} over Σ , two theories \mathcal{T} and \mathcal{T}' are $\{\Sigma, \mathcal{I}\}$ -equivalent if for each model M of \mathcal{T} that extends \mathcal{I} , its restriction to Σ can be extended to a model of \mathcal{T}' extending \mathcal{I} , and vice-versa and the extensions are unique.

A formula is in Negation Normal Form (NNF) if \neg only occurs directly in front of atoms and if conjunctions and disjunctions are in left-associative form (e.g., $A \vee (B \vee (\dots))$). We assume, without loss of generality, that our sentences and rule bodies are NNF.

III. GROUNDING TO PARAMETRISED GROUND FO(\cdot)

This section describes an algorithm to construct the grounding of a theory \mathcal{T}_{in} over Σ in the context of a 3-valued, consistent interpretation \mathcal{I}_{in} . The algorithm transforms \mathcal{T}_{in} to a $\{\Sigma, \mathcal{I}\}$ -equivalent ground—quantifier-free— theory \mathcal{T}_g and a “mapping” theory \mathcal{T}_m consisting of explicit definitions for symbols of Σ that were eliminated from \mathcal{T}_g .

The algorithm takes as parameter a set ResF of “residual” function symbols, function symbols allowed in \mathcal{T}_g . In our algorithm, functions f/n not in ResF are replaced by their “graph” predicate symbol $g_f/n+1$. If ResF is empty, then all atoms in the grounding will be domain atoms; by translating these into propositional symbols, such a theory can be mapped into an “equivalent” propositional theory.

The grounding process is described as two stratified sequences of $\{\Sigma, \mathcal{I}\}$ -equivalence preserving rewrite rules, rewriting the theories \mathcal{T}_g and \mathcal{T}_m . Theory \mathcal{T}_g is initialized as \mathcal{T}_{in} , \mathcal{T}_m as the empty set. The rewrite rules operate on \mathcal{T}_g , substituting expressions or rules by simpler ones, and sometimes introducing new definitions to \mathcal{T}_g or \mathcal{T}_m . E.g. $\neg\neg\varphi \mapsto \varphi$ is the rule that replaces occurrences of $\neg\neg\varphi$ in \mathcal{T}_g by φ .

A. Phase 1: simplifying the syntax

The first phase consists of iterated rewriting of \mathcal{T}_g by the rewrite rules specified below. The rewriting process terminates when no more rules are applicable.

- $\varphi \Leftrightarrow \psi \mapsto \varphi \Rightarrow \psi \wedge \psi \Rightarrow \varphi$.
- $\varphi \Rightarrow \psi \mapsto \neg\varphi \vee \psi$.
- $\neg\neg\varphi \mapsto \varphi$.
- $\neg(\varphi \vee \psi) \mapsto \neg\varphi \wedge \neg\psi$.
- $\neg(\varphi \wedge \psi) \mapsto \neg\varphi \vee \neg\psi$.
- $\neg(\forall x : \varphi) \mapsto \exists x : \neg\varphi$.
- $\neg(\exists x : \varphi) \mapsto \forall x : \neg\varphi$.

- $\neg(t \sim t') \mapsto t \not\sim t'$. We use \sim to denote a comparison operator such as $\leq, <, =, \neq, \dots$ and $\not\sim$ denotes respectively $>, \geq, \neq, =, \dots$.
- $(\varphi \wedge \psi) \wedge \gamma \mapsto \varphi \wedge (\psi \wedge \gamma)$.
- $(\varphi \vee \psi) \vee \gamma \mapsto \varphi \vee (\psi \vee \gamma)$.
- **Unnest** function terms $f(\bar{t}) A[f(\bar{t})] \mapsto \exists x : f(\bar{t}) = x \wedge A[f(\bar{t})/x]$ where A is an occurrence of an atom in an FO sentence or rule body and A is not of the form $f(\bar{t}) = t$.
 $A[f(\bar{t})] \leftarrow \varphi \mapsto \forall y : A[f(\bar{t})/y] \leftarrow f(\bar{t}) = y \wedge \varphi$.
 $\{\bar{x} : \varphi : t[f(\bar{t})]\} \mapsto \{\bar{x}, y : \varphi \wedge f(\bar{t}) = y : t[y]\}$.

These rewrite rules eliminate \Leftrightarrow and \Rightarrow , drive negation deeper and flatten conjunctions and disjunctions. In the resulting theory, negation is in front of atoms of user-defined symbols. All occurrences of function symbols $f \notin \text{ResF}$ are top left symbols in equalities $f(\bar{t}) = t$. Note, if ResF is empty, such atoms are of the form $f(\bar{t}) = t$ with t_1, \dots, t_n, t either domain elements (e.g., natural numbers) or variables. As final step in this phase, function symbols are replaced by their graph as follows. For each function symbol $f/n \notin \text{ResF}$, we introduce a new predicate symbol $g_f/n+1$, apply the rewrite rule $f(\bar{t}) = t \mapsto g_f(\bar{t}, t)$, add $\forall \bar{x} : \#(\{y : g_f(\bar{x}, y) : 1\}) = 1$ to \mathcal{T}_g and add $\forall \bar{x} y : f(\bar{x}) = y \Leftrightarrow g_f(\bar{x}, y)$ to \mathcal{T}_m .

B. Phase 2: Grounding

From now on, all expressions explicitate the domains of variables, written as $\forall \bar{x} \in \bar{D} : \varphi$ or $\{\bar{x} \in \bar{D} : \varphi : t\}$. Initially, \bar{D} is $\tau_1^{\mathcal{I}_{in}} \times \dots \times \tau_n^{\mathcal{I}_{in}}$, where τ_i is the type of x_i ³.

The second phase applies the following set of rewrite rules and also terminates when no more rules are applicable.

- **Split conjunctive sentences:** $\varphi \wedge \psi \mapsto \varphi, \psi$ where $\varphi \wedge \psi$ is a sentence in \mathcal{T}_g .
- **Instantiate**, for some $\bar{d} \in \bar{D}$:
 $\forall \bar{x} \in \bar{D} : \psi \mapsto \psi[\bar{x}/\bar{d}] \wedge \forall \bar{x} \in \bar{D} - \bar{d} : \psi$.
 $\exists \bar{x} \in \bar{D} : \psi \mapsto \psi[\bar{x}/\bar{d}] \vee \exists \bar{x} \in \bar{D} - \bar{d} : \psi$.
 $\forall \bar{x} \in \bar{D} : A \leftarrow \psi \mapsto A[\bar{x}/\bar{d}] \leftarrow \psi[\bar{x}/\bar{d}]$
 $\forall \bar{x} \in \bar{D} - \bar{d} : A \leftarrow \psi$.
 $\{\bar{x} \in \bar{D} : \varphi : t\} \mapsto \{\varphi[\bar{x}/\bar{d}] : t[\bar{x}/\bar{d}]\} \cup \{\bar{x} \in \bar{D} - \bar{d} : \varphi : t\}$.
- **Evaluate** $t \mapsto t^{\mathcal{I}_{in}}$ if t is a domain term with a single value in $t^{\mathcal{I}_{in}}$. $P(\bar{d}) \mapsto P(\bar{d})^{\mathcal{I}_{in}}$, if $P(\bar{d})^{\mathcal{I}_{in}} \neq \mathbf{u}$. This rule is not applied to occurrences of $P(\bar{d})$ in definitions that define P ⁴.
- **Introduce Tseitin** $\psi \mapsto T_\psi$, where ψ is an occurrence of a formula without free variables in \mathcal{T}_g and T_ψ is a newly introduced propositional symbol. Additionally, if ψ occurs in a definition Δ , the rule $T_\psi \leftarrow \psi$ is added to Δ , otherwise, the singleton definition $\{T_\psi \leftarrow \psi\}$ is added to \mathcal{T}_g . The rule is not applied if ψ is a domain literal, sentence or rule body.
- **Introduce term** $t \mapsto c_t$, where t is an occurrence of a term without free variables in \mathcal{T}_g and c_t is a newly

³Recall that \mathcal{I}_{in} specifies a finite domain $\tau^{\mathcal{I}_{in}}$ for every type τ .

⁴By definition, **Evaluate** checks well-typedness of expressions.

introduced constant over the type of t . Additionally, $t = c_t$ is added to \mathcal{T}_g . The rule is not applied if t is a domain element or occurs in an atom of the form $P(\bar{e})$, $f(\bar{e}) \sim e_0$ or $agg(\{L_1 : e_1\} \cup \dots \cup \{L_n : e_n\}) \sim e_0$.

- **Simplify**

$$\begin{array}{ll} \neg \mathbf{t} \mapsto \mathbf{f} & \neg \mathbf{f} \mapsto \mathbf{t} \\ \psi \vee \mathbf{t} \mapsto \mathbf{t} & \psi \wedge \mathbf{f} \mapsto \mathbf{f} \\ \psi \vee \mathbf{f} \mapsto \psi & \psi \wedge \mathbf{t} \mapsto \psi. \\ \forall \bar{x} \in \bar{D} : \mathbf{t} \mapsto \mathbf{t} & \exists \bar{x} \in \bar{D} : \mathbf{t} \mapsto \mathbf{t} \\ \forall \bar{x} \in \bar{D} : \mathbf{f} \mapsto \mathbf{f} & \exists \bar{x} \in \bar{D} : \mathbf{f} \mapsto \mathbf{f} \\ \forall \bar{x} \in \emptyset : \psi \mapsto \mathbf{t} & \exists \bar{x} \in \emptyset : \psi \mapsto \mathbf{f} \\ \{\bar{x} \in \emptyset : \psi : t\} \mapsto \{\mathbf{f} : t\} & \{\bar{x} \in \bar{D} : \mathbf{f} : t\} \mapsto \{\mathbf{f} : 0\} \end{array}$$

After application of the above rewrite rules, we obtain a theory in Ground Normal Form (GNF).

Definition III.1. An FO(\cdot) theory \mathcal{T} is in *Ground Normal Form (GNF)* if all its sentences and rules are of one the following forms (with all L_i 's domain literals):

$$\begin{array}{l} L_1 \vee \dots \vee L_n. \quad Q(\bar{e}). \quad f(\bar{e}) \sim e_0. \\ agg(\{L_1 : e_1\} \cup \dots \cup \{L_n : e_n\}) \sim e_0. \\ P(\bar{e}) \leftarrow L_1 \wedge \dots \wedge L_n. \\ P(\bar{e}) \leftarrow L_1 \vee \dots \vee L_n. \\ P(\bar{e}) \leftarrow Q(\bar{e}'). \quad P(\bar{e}) \leftarrow f(\bar{e}) \sim e_0. \\ P(\bar{e}) \leftarrow agg(\{L_1 : e_1\} \cup \dots \cup \{L_n : e_n\}) \sim e_0. \end{array}$$

Theorem III.2. For input \mathcal{T}_{in} , \mathcal{I}_{in} and ResF, let \mathcal{T}_g and \mathcal{T}_m be the computed theories at any time during the rewrite process. Then \mathcal{T}_{in} and $\mathcal{T}_g \cup \mathcal{T}_m$ are $\{\Sigma, \mathcal{I}\}$ -equivalent. The rewrite-process terminates and the resulting theory \mathcal{T}_g is in GNF and contains only function symbols in ResF.

The equivalence follows from the fact that each rewrite rule preserves $\{\Sigma, \mathcal{I}\}$ -equivalence. That the resulting theory is in GNF follows from the fact that none of the rewrite rules apply in the context of a GNF theory and that at least one rewrite rule is applicable to any theory not in GNF.

Termination of phase 1 is straightforward. To prove termination of phase 2, it can be shown that a well-founded order exists on theories for the presented rewrite rules. This order depends among others on the nesting depth of symbols, the nesting and domain size of quantifications and the number of occurrences of symbols in the theory. The formal presentation of the well-founded order is out of the scope of this paper.

C. Concrete grounding algorithm

The rewrite process of the previous section is not confluent. By imposing different rewrite strategies, it can be instantiated to a class of –sound– grounding algorithms. To obtain a state-of-the-art grounding algorithm, one should select an instantiation that minimizes the number of traversals through formulas in search for applicable rewrite rules, the memory and time complexity of the algorithm, the grounding size, The rewrite strategy that is implemented in our system is quite complex and a full presentation is out of the scope of this paper; we highlight the most important considerations here:

- The top priority is to minimize the grounding size, followed by minimizing the running time and memory usage of the grounding algorithm.

- **Instantiate** is performed top-down and depth-first. This allows to simplify formulas early and reduces the memory overhead of storing partial results.
- **Simplify** and **Evaluate** are applied eagerly, as they may considerably reduce the size of formulas.
- The number of introduced symbols should be minimized. E.g., it is important to avoid the creation of multiple Tseitin symbols for multiple occurrences of the same term, atom or formula.

An important optimisation is to first make \mathcal{I}_{in} more precise by applying *symbolic propagation* for \mathcal{T}_{in} to it. This leads to a more precise 3-valued structure \mathcal{I}_{in}' that approximates all instances of \mathcal{I}_{in} that are models of \mathcal{T}_{in} . This process was described first in [34] and later for the restriction to FO in [31] where it was called *lifted unit propagation*, as this propagation is indeed a symbolic version of unit propagation. With the refined structure \mathcal{I}_{in}' , the ground theories are sometimes orders of magnitude smaller than w.r.t. \mathcal{I}_{in} [35].

IV. MODEL EXPANSION FOR GENERAL GROUND FO(\cdot)

In this section, we present an MX algorithm which takes as input a general ground FO(\cdot) theory \mathcal{T}_g in GNF and a 3-valued input structure \mathcal{I}_{in} .⁵ As before, we assume that all types τ are interpreted as finite sets $\tau^{\mathcal{I}_{in}}$ of domain elements.

The algorithm is based an existing MX algorithm for function-free GNF, implemented in the system MINISAT(ID) and described in [22][21]. That algorithm is a conflict-driven clause-learning (CDCL) search algorithm, extended to handle inductive definitions and aggregates. Recall that function-free GNF can be obtained by running the algorithm of the previous section with ResF = \emptyset . The algorithm developed here uses a generalisation of the technique of lazy clause generation[30] to support full GNF, explained later in this section.

A. Adapt existing CDCL algorithm to our setting

The state of the algorithm consists of a theory \mathcal{T}_s , and a three-valued interpretation \mathcal{I} . We present \mathcal{I} as the sequence of its true literals, ordered by the time at which the literals were derived. A literal L_i in this sequence is annotated L_i^D if it is a decision literal; other literals were derived by propagation. Initially, \mathcal{T}_s is the input theory \mathcal{T}_g and \mathcal{I} is the empty set. For ease of presentation, we use a slight adaptation of GNF in the rest of the paper: any sentence A , with A one of the atoms $Q(\bar{e})$, $f(\bar{e}) \sim e_0$ or $agg(\{L_1 : e_1\} \cup \dots \cup \{L_n : e_n\}) \sim e_0$, is generalized as an equivalence $P(\bar{d}) \Leftrightarrow A$. Any such sentence A in \mathcal{T}_g is then added as the sentence $\mathbf{t} \Leftrightarrow A$ to \mathcal{T}_s .

As an initial step of the algorithm, definitions Δ in \mathcal{T}_s are simplified. If Δ is not recursive (or if it can be stratified), it can be split in a set of subdefinitions $\Delta_1, \dots, \Delta_n$ as shown in [9]. These are added to \mathcal{T}_s and Δ is removed from it.

A number of inference rules operate on such states. The first four rules describe a basic CDCL SAT-solver: **Decide**: Select non-deterministically a domain literal L such that $L^{\mathcal{I}} = \mathbf{u}$, and append L^D to \mathcal{I} . **UP**: Apply unit propagation to a

⁵The theory \mathcal{T}_s computed during grounding contains only explicit definitions of symbols that do not occur in \mathcal{T}_g and can be ignored during search.

clause in \mathcal{T}_s and append the derived literal L to \mathcal{I} . **Fail:** If \mathcal{I} is inconsistent and contains no decision literals, the algorithm returns “unsatisfiable”. **Learn:** If \mathcal{I} is inconsistent and contains decision literals, conflict-driven clause-learning is applied to \mathcal{I} and \mathcal{T}_s to construct a learnt clause C which is added to \mathcal{T}_s . Backjumping to the level of the second youngest literal of this clause occurs. The output of the algorithm is either **fail** or a three-valued interpretation \mathcal{I} expanding \mathcal{I}_{in} such that every more precise two-valued interpretation \mathcal{I}' is a model of \mathcal{T}_g .

The remaining propagation rules, presented in the next section, then serve to perform propagation on the non-clausal components of \mathcal{T}_s . In the MINISAT(ID) algorithm, this consists of four additional rules, **Aggregate**, **Completion**, **Unfounded** and **Wellfounded**. The first checks for propagation over aggregate expressions by reasoning on the bounds of the aggregate function (the minimum and maximum value the function can still take in a partial structure). The latter two rules apply to inductive definitions. The rule **Completion** is only executed in the initial phase; it applies to a definition Δ and adds its completion to \mathcal{T}_s . If Δ is equivalent with its completion (for example for Tseitin symbols introduced only in sentences), Δ can be dropped from \mathcal{T}_s , as shown in [9]. **Unfounded** searches for unfounded sets[33] in a definition Δ and if an unfounded set U is found, propagates all its atoms as **f** (i.e., it appends $\neg U$ to \mathcal{I}). When \mathcal{I} is a 2-valued interpretation, **Wellfounded** checks if \mathcal{I} is a well-founded model of a definition Δ , as shown in [32]. In what follows, these rules will be extended (and new ones will be added), to handle the more general format of GNF.

We omit a discussion on CDCL improvements such as the 2-watched literal scheme and restarts; they can be incorporated straightforwardly in the presented algorithm. The experimental evaluation is based on a state-of-the-art CDCL algorithm.

B. Approach to extend to GNF

Any GNF theory can be transformed into a $\{\Sigma, \mathcal{I}\}$ -equivalent function-free GNF theory. In that case, the inference rules presented above are sufficient for a complete algorithm. One approach to obtain such a theory was already presented in Section III: to apply the rewrite algorithm with an empty set ResF. However, instead of generating such a function-free theory *eagerly*, before search, in the rest of the section we present a concrete algorithm to generate such a function-free theory *lazily* (i.e. during search). The algorithm is based on the technique of *lazy clause generation*, presented in [30]. Lazy clause generation alleviates the blowup of creating the full function-free ground theory in advance in two ways: first it uses smarter technique than graphing functions, and second, it only generates these clauses when they would contribute to the search, i.e. on the moment that they would propagate. We generalise the scheme by not only lazily generating clauses but lazily generating GNF sentences. To avoid infinite loops, we impose the following partial order on GNF expressions; only smaller expressions can be generated (w.r.t. this ordering).

$$\begin{aligned} & \text{definitional rules} \ll \text{reified aggregates} \ll P \Leftrightarrow Q(\bar{e}) \\ & \ll P \Leftrightarrow f(\bar{e}) \sim e_0 \ll P \Leftrightarrow f(\bar{d}) \sim e_0 \ll \text{clauses} \end{aligned}$$

We assume \bar{e} contains at least one domain terms (distinguishing it from the case $P \Leftrightarrow f(\bar{d}) \sim e_0$); additionally, we

assume expressions over the operator \leq are ordered lower below similar expressions over other comparison operators.

In the rest of the section, we show how the various GNF expressions that possibly contain function terms are supported. For each of these, the presentation consists of three components. First, a set of (non-ground) sentences of the form $\forall \bar{x} : \varphi \Rightarrow L$; intuitively, these will be the set of propagations or decompositions we consider for the expression at hand, propagating the right-hand side (the *head*) when the left-hand side (the *body*) is true. Second, a discussion on how to quickly find instances of \bar{x} for which φ holds in \mathcal{I} and $L^{\mathcal{I}}$ is not true. The algorithm then consists of adding the sentence $\varphi[\bar{x}/\bar{d}] \Rightarrow L[\bar{x}/\bar{d}]$ for the relevant instantiations \bar{d} of \bar{x} . Third, a discussion on when such derived sentences will be added to \mathcal{T}_s , which will depend on the expression at hand. As discussed previously, the type of the derived sentences should be ordered below the type of the original expression.

C. Encoding functions

To handle constraints over functions f/n with domain D in a solver that decides on domain atoms, we use the *range encoding*[30]. A domain term $t = f(\bar{d})$ with f a function mapping to the domain $D = \{d_1, \dots, d_n\}$ is encoded as the set of propositional symbols $\{T_{t \leq d_1}, \dots, T_{t \leq d_n}\}$. For each t , we define $\min_t = \max\{d \in D \mid T_{t \leq \text{prev}(d)} = \mathbf{f}\}$ and $\max_t = \min\{d_i \in D \mid T_{t \leq d_i} = \mathbf{t}\}$. The range of t is then defined as $[\min_t, \max_t]$. The values \min_t and \max_t can be computed from \mathcal{I} , but an efficient algorithm should store them and adapt them incrementally whenever \mathcal{I} changes.

The range encoding is selected over encoding the function as a set of equalities $T_{t=d_i}$ as the encoding of inequalities is more compact (and the encoding for equalities is only a factor 2 larger) and choices on encoding atoms more often eliminate subsets of the domain instead of just one value. A more in-depth comparison is provided in [30].

In the sequel, for domain term t and value d , we use $[t \leq d]$ to denote the atom $T_{t \leq d}$ if $d \in D = \{d_1, \dots, d_n\}$, the atom **f** if $d < d_1$, the atom **t** if $d > d_n$ and otherwise the atom $T_{t \leq d'}$, with d' the smallest domain element in D larger than d . All other comparison operators \sim can be defined in terms of \leq . We use $[t \sim d]$ as a shorthand for those rewritings. E.g. $[t \neq d_i]$ denotes $[t \leq d_{i-1}] \vee \neg [t \leq d_i]$.

The following set of non-ground clauses represents the dependencies between those symbols.

$$\begin{aligned} & [t \leq d_n] \\ & \forall x \in D - d_n : [t \leq x] \Rightarrow [t \leq \text{next}(x)]. \\ & \forall x \in D - d_1 : [t > x] \Rightarrow [t > \text{prev}(x)]. \end{aligned}$$

The propagation rule **Encode** is applied to a domain term t the first time it appears in \mathcal{T}_s , and it adds the grounding of the above formulas to \mathcal{T}_s . For small domains D ($\|D\| < 100$), this is done eagerly; for larger ones this is done lazily as described in [30]. We do not elaborate the details here. Additionally, to take care of interpreting f when we have a model of the encoding clauses, **Encode** adds the mapping sentence $\forall x \in D : [f(\bar{d}) = x] \Rightarrow f(\bar{d}) = x$ to \mathcal{T}_m .

Example IV.1. Consider the theory \mathcal{T}_g consisting only of the sentence $P \Leftrightarrow f(1) \leq 3$, with f typed as $f(\tau) : \tau'$,

τ interpreted as D , τ as D' . **Encode** will then add the grounding of the above sentences for $t = f(1)$. It does not add instantiations for any other term $f(d)$, $d \neq 1$, which has an important impact if D is large. In this case, the result of MX is a three-valued interpretation of which any two-valued extension is a model of the theory. For example, interpretation $\mathcal{I} = \{P, f(1) = 3\}$ contains enough information: all structures more precise than \mathcal{I} , are models of \mathcal{T}_g .

D. Comparison constraint

The propagation rule **Comparison** applies to constraints $P \Leftrightarrow c \leq c'$, with P a domain atom and c and c' domain terms over a domains D , respectively D' . The propagations we consider can be represented as the following sentences.

$$\begin{aligned} \forall x \in D \cup D' : [c \leq x] \wedge [c' \geq x] &\Rightarrow P. \\ \forall x \in D \cup D' : [c > x] \wedge [c' < x] &\Rightarrow \neg P. \\ \forall x \in D : [c' \leq x] \wedge P &\Rightarrow [c \leq x]. \\ \forall x \in D : [c' \geq x] \wedge \neg P &\Rightarrow [c > x]. \\ \forall x \in D' : [c \geq x] \wedge P &\Rightarrow [c' \geq x]. \\ \forall x \in D' : [c \leq x] \wedge \neg P &\Rightarrow [c' < x]. \end{aligned}$$

It is easy to see that together with the encoding of c and c' , this set of sentences is $\{\Sigma, \mathcal{I}\}$ -equivalent to the original constraint. Comparison constraints over comparison operators other than \leq are converted into 1 or 2 comparison constraints over \leq (with Tseitin introduction in the latter case).

Instantiations are generated as follows. **Comparison** checks for each of the non-ground sentences whether the body is true, but only for instantiations of x with $\min_c, \max_c, \min_{c'}$ and $\max_{c'}$. This is checked whenever one of those values increases (for min) or decreases (for max) and whenever P becomes assigned. It is straightforward to show that this is sufficient, i.e., when **UP**, **Encode** and **Comparison** are at fixpoint (without conflict), none of the above sentences has a true body and an unknown or false head for any instantiation.

Example IV.2. Consider a constraint $P \Leftrightarrow c \leq c'$, with c a range of $[3, 10]$, c' a range of $[7, 20]$ and P is true in \mathcal{I} . When \mathcal{I} is extended with $[c \geq 8]$ to \mathcal{I}' , **Comparison** checks for $x = 8$ which of the left-hand sides are true, which is the case for the sentence $[c \geq x] \wedge P \Rightarrow [c' \geq x]$. As the head is not true in \mathcal{I}' , the sentence is added to \mathcal{T}_s (in clausal form) and **UP** will derive $[c' \geq 8]$.

E. Aggregates

Next, we introduce propagation rules for sentences of the form $P \Leftrightarrow \text{agg}(\{L_1 : e_1\} \cup \dots \cup \{L_n : e_n\}) \leq e_0$ where agg is either a maximum or sum aggregate function. As above, other comparison operators can be rewritten into constraints over \leq . Cardinality constraints are rewritten straightforwardly into sum constraints and minimum into maximum constraints. The rules for product aggregates are not presented here, as they are similar to those for sum (although complicated by the non-monotonicity of product for terms with negative values).

The rule **Encode_{max}** rewrites a maximum constraint $P \Leftrightarrow \max(S) \leq c$ into the following sentences

$$\begin{aligned} P \wedge L_i &\Rightarrow e_i \leq e_0 && \text{for each } i \in [1, n] \\ \neg P &\Rightarrow \bigvee_{i \in [1, n]} (L_i \wedge c_i > e_0) \end{aligned}$$

As the rewriting consists of only $n + 1$ ground sentences, it is done eagerly for any maximum aggregate constraint in \mathcal{T}_s .

Enumerating the clauses generated from a sum constraint, by the **Encode_{sum}** propagation rule, is out of the scope of this paper, we only give an example: the sentence

$$\left[\sum_{i | L_i^{\mathcal{I}} \neq \mathbf{f}} \max_{e_i} \leq e_0 \right] \bigwedge_{i | L_i^{\mathcal{I}} = \mathbf{f}} \neg L_i \Rightarrow P$$

is in fact a smart instantiation of the clause

$$\forall \bar{x} : \left(\left[\sum_i x_i \leq e_0 \right] \wedge \bigwedge_{i | L_i^{\mathcal{I}} \neq \mathbf{f}} [e_i \leq x_i] \bigwedge_{i | L_i^{\mathcal{I}} = \mathbf{f}} \neg L_i \right) \Rightarrow P$$

This sentence expresses that P is true if the sum of the maxima of all terms in S , of which the condition is not false, is lower or equal than e_0 . The other sentences are similar in idea, but not presented here. Similarly to handling comparison constraints, propagation is checked for the bounds of all terms and for all assignments to the associated atoms.

F. General ground atoms

Constraints of the form $P \Leftrightarrow q(\bar{e})$ and $P \Leftrightarrow f(\bar{e}) \sim e_0$ are handled by waiting until all domain terms in \bar{e} are assigned. At that moment, the instantiated constraint is generated, which coincides with instantiations of the sentence

$$\begin{aligned} \forall \bar{x} \in \text{dom}_{\bar{e}} : [\bar{e} = \bar{x}] &\Rightarrow (P \Leftrightarrow Q(\bar{x})), \text{ respectively} \\ \forall \bar{x} \in \text{dom}_{\bar{e}} : [\bar{e} = \bar{x}] &\Rightarrow (P \Leftrightarrow f(\bar{x}) \sim e_0). \end{aligned}$$

The propagation rule **Encode_{general}** adds the above sentences whenever the value of each of the c_i is known (applying Tseitin introduction to generate sentences in GNF). The former results in a set of clauses, the latter in a clause and a comparison constraints, which are both constraints of a lower type than the original sentence.

Constraints of the form $P \Leftrightarrow f(\bar{e}) \sim e_0$ are in fact a generalisation of the *element* constraint from the field of Constraint Programming[17], as the next example shows.

Example IV.3. An element constraint $\text{element}(c, A, i)$ expresses that a cp-variable (or constant) c takes the value at index i of array A . It is well-known that an array is in fact a function f_A from indices to values. The element constraint can then be modelled as the sentence $f_A(i) = c$ and handled lazily as described above, by generating the comparison constraint $f_A(d) = c$ when i is assigned to d in \mathcal{I} . It is possible that A (f_A) is very large or not completely known in advance.

Obviously, it is sometimes possible to derive propagation even before \bar{e} is completely instantiated. Investigating the benefit of such propagation is part of future work.

G. Definitions with function terms

In the standard case (no function terms), definitions are handled by applying the rules **Completion**, **Unfounded** and **Wellfounded**. Definitions containing function terms should be handled carefully, for which we introduce the extended rules **Completion'**, **Unfounded'** and **Wellfounded'**.

Consider a definition Δ defining, among others, the symbol P by the rules $\{P(\bar{e}_1) \leftarrow \varphi_1, \dots, P(\bar{e}_n) \leftarrow \varphi_n\}$. The completion of P for Δ is the (non-ground) sentence

$$\forall \bar{x} : P(\bar{x}) \Leftrightarrow \left(\bigvee_{i \in [1, n]} [\bar{x} = \bar{e}_i] \wedge \varphi_i \right)$$

The rule **Completion'** adds the equivalent sentences

$$\begin{aligned} \bigwedge_{i \in [1, n]} \varphi_i &\Rightarrow P(\bar{e}_i) \\ \forall \bar{x} : P(\bar{x}) &\Rightarrow \left(\bigvee_{i \in [1, n]} [\bar{x} = \bar{e}_i] \wedge \varphi_i \right) \end{aligned}$$

The former sentence is added eagerly for each i (as it is already ground). For the latter sentence, **Completion'** adds its instantiation of \bar{x} with \bar{d} to \mathcal{T}_s for atoms $P(\bar{d})$ true in \mathcal{I} .

An issue with the condition on instantiation is that propagations might be missed. Indeed, the latter sentence of **Completion'** is only instantiated for $P(\bar{d})$ true in \mathcal{I} ; however, if $\left(\bigvee_{i \in [1, n]} [\bar{d} = \bar{e}_i] \wedge \varphi_i \right)$ is false, then $\neg P(\bar{d})$ is entailed. If $P(\bar{d})$ does not occur in \mathcal{T}_s (and is never added by other rules), it will not be decided, resulting in an interpretation of which not all two-valued extensions are models. It is easy to show that in a (non-failed) state in which no more inference rules are applicable, all unassigned domain atoms over defined symbols have to be false. Extending the interpretation in this way, denoted as the rule **Defined-false**, then restores the post-condition of the algorithm.

For **Unfounded'** and **Wellfounded'**, we take an approach similar to previous sections: both rules are only applied when all domain terms occurring in Δ are assigned. In such situations, replacing all domain terms in Δ with their interpretation results in a definition to which the existing propagation rules **Unfounded** and **Wellfounded** can be applied. If one of these generates an explanation clause C , this clause is only valid conditionally, as we had to substitute several constants in order to obtain it. So instead of adding C to \mathcal{T}_s , we add

$$\left(\bigwedge_{c|c \text{ occurs in } \Delta} [c = c^{\mathcal{I}}] \right) \Rightarrow C.$$

Example IV.4. Consider part of a graph application consisting of a function `next` mapping nodes to nodes and a constant `start` of type node. Suppose the aim is to compute a relation r on nodes, representing all nodes reachable from the start node through `next`. The following is a definition of r ⁶.

$$\left\{ r(\text{start}). \quad \forall x : r(\text{next}(x)) \leftarrow r(x). \right\}$$

In the context of an interpretation \mathcal{I} over domain $\{a, b, c\}$, with $\text{start}^{\mathcal{I}} = a$ and $\text{next}^{\mathcal{I}} = \{a \mapsto b, b \mapsto a, c \mapsto c\}$, the definition reduces to the following definition, to which **Unfounded** can be applied:

$$\left\{ r(a). \quad r(b) \leftarrow r(a). \quad r(a) \leftarrow r(b). \quad r(c) \leftarrow r(c). \right\}$$

Unfounded would then derive the unit clause $\neg r(c)$; consequently, **Unfounded'** generates the clause

$$\begin{aligned} ([\text{start} = a] \wedge [\text{next}(a) = b] \\ \wedge [\text{next}(b) = a] \wedge [\text{next}(c) = c]) &\Rightarrow \neg r(c). \end{aligned}$$

⁶Note that the size of the grounding of this definition is linear in the size of the domain, instead of quadratic if functions would be graphed.

H. Pre-interpretation over some symbols

As discussed in the section on grounding, next to the ground theory, the algorithm maintains a partial, symbolic interpretation \mathcal{I}_{in} which is guaranteed to be consistent. However, we do not want to add this interpretation as constraints to the theory, for the same reason as we do not want to eagerly generate the full propositional grounding; e.g., if very few atoms over an interpreted predicate occur in \mathcal{T}_s . Instead, the following propagation rule takes care of adding just enough of \mathcal{I}_{in} to obtain interpretations that are consistent with \mathcal{I}_{in} . Rule **check- \mathcal{I}_{in}** adds a clause $(\neg)A$ to \mathcal{T}_s for every atom A in \mathcal{T}_s such that $A^{\mathcal{I}_{in}} = \mathbf{t}(\mathbf{f})$ and $(\neg)A \notin \mathcal{I}$.⁷

Example IV.5. Consider a theory \mathcal{T}_s with constraint $P(c) \vee \neg P(c')$, with P over a large domain D and interpreted in \mathcal{I}_{in} . Adding clauses $(\neg)P(d)$ for every domain-element $d \in D$ would cause an immense grounding. However, lazily adding this whenever a value for c or c' is chosen, results in a theory where only the relevant literals are asserted.

I. Complete search algorithm

Next to the set of inference rules, a search algorithm consists of an execution order \ll on its rules. For the MINISAT(ID) algorithm, the order is **Fail** \ll **Learn** \ll **UP** \ll **Unfounded** \ll **Decide** \ll **Wellfounded**. This order was chosen with efficiency in mind. E.g., whenever **Fail** is possible, it is useless to propagate further; **UP** is preferred over **Unfounded** because it is much cheaper and often derives more propagation; etc.

An additional concern is to not generate the same expression lazily multiple times, preferably without having to explicitly keep track of this. The approach taken is to order the rules in the inverse order of the type of constraints they apply to. This results in the following priority order (recall that **Encode_{max}** is executed in the initial phase).

$$\begin{aligned} &\mathbf{Fail} \ll \mathbf{Learn} \ll \mathbf{UP} \ll \mathbf{check-}\mathcal{I}_{in} \\ &\ll \mathbf{Encode} \ll \mathbf{Comparison} \ll \mathbf{Completion}' \\ &\ll \mathbf{Encode}_{\text{sum}} \ll \mathbf{Encode}_{\text{general}} \ll \mathbf{Unfounded}' \\ &\ll \mathbf{Decide} \ll \mathbf{Wellfounded}' \ll \mathbf{Defined-false} \end{aligned}$$

Theorem IV.6 (Soundness and completeness). *For any GNF theory \mathcal{T} and consistent interpretation \mathcal{I}_{in} over $\Sigma(\mathcal{T})$, the algorithm terminates and returns an interpretation \mathcal{I} , consistent with \mathcal{I}_{in} , such that all two-valued extensions of $\mathcal{I} \cup \mathcal{I}_{in}$ are models of \mathcal{T} , or **fail** if no models of \mathcal{T} exist that extend \mathcal{I}_{in} .*

The proof is omitted due to lack of space.

V. EXPERIMENTS

The grounding algorithm is implemented in the IDP³ system[4], a knowledge-base system supporting state-of-the-art model expansion, as can be observed from previous ASP competitions [10], [5]⁸. The search algorithm is implemented in the solver CONSTRAINT-(ID), extending the state-of-the-art algorithm MINISAT(ID) [7]. As benchmarks, we used the benchmarks of the fourth ASP competition⁹ in the NP

⁷By definition, **check- \mathcal{I}_{in}** checks well-typedness of expressions.

⁸Results of the fourth ASP competition are not available as of this writing.

⁹Available at <https://www.mat.unical.it/aspcomp2013/OfficialProblemSuite>

Benchmark	# inst.	# solved	avg. time(sec)	avg. size (# atoms)
bottle fill.	12	12(12)	39.5(39.3)	5 * 10⁵ (7 * 10 ⁵)
crossing min.	20	7(7)	372.3(340.5)	6 * 10³ (3 * 10 ⁵)
graceful graphs	17	11 (3)	387.1(244.5)	1 * 10⁶ (3 * 10 ⁷)
incr. sched.	20	20 (0)	0.8(— —)	1 * 10⁴ (— —)
no-mystery	8	8(8)	2.9(0.8)	1 * 10 ⁵ (1 * 10 ⁵)
pattern matching	6	6(6)	0.4 (58.2)	2 * 10⁴ (2 * 10 ⁷)
ricochet robots	8	5(5)	267.7(269.4)	9 * 10 ⁶ (9 * 10 ⁶)
sokoban	7	7(7)	9.9(4.5)	5 * 10⁴ (6 * 10 ⁴)
solitaire	20	14(14)	28.1(32.2)	3 * 10 ⁴ (3 * 10 ⁴)
stable marriage	20	20(20)	68.5(66.3)	2 * 10 ⁷ (2 * 10 ⁷)
still life	7	4(4)	19.9(20.2)	2 * 10³ (6 * 10 ³)
valve location	5	5(5)	5.8 (10.8)	1 * 10⁶ (2 * 10 ⁶)
disj. scheduling	20	20 (5)	2.0 (27.5)	3 * 10³ (2 * 10 ⁷)
packing	20	20 (4)	0.7 (279.0)	1 * 10⁴ (6 * 10 ⁷)

TABLE I. EXPERIMENTAL EVALUATION

complexity class. Also, we included two classic constraint programming problems, present at the third ASP competition: the *disjunctive scheduling* problem and the *packing* problem. Problem instances were taken from the fourth ASP competition (at the time of writing, only a limited subset has already been disclosed) and the third ASP competition. The search implementation is currently limited to functions over integer domains, which is not an issue in the considered benchmarks.

Each of the benchmarks and associated instances was solved using IDP³, measuring performance and size of the grounding¹⁰. Two different setups were used. The *gnf* setup applies MX with as ResF, the set of all functions with integer domains. The *gnf* setup, able to apply all ideas presented in this paper, is compared to the (function-free) reference setup *ref*, which uses ResF = ∅. As discussed earlier, this results in an effectively propositional grounding, in which case CONSTRAINT-(ID) collapses to MINISAT-(ID).

In table I, we report on the performance per benchmark, measured as the number of solved instances, the average total time for the *solved* instances, and the average size of the grounding; bold numbers indicate clear winners. For each, we report the results as $x(y)$, with x the result of the *gnf* setup and y the result of the reference setup. The time limit was 1000 seconds, the memory limit 3.5 GB.

The most important conclusion this table gives is that the techniques described in this paper are crucial to solve some problems: *incremental scheduling* could not be solved by *ref*, but posed no problem to *gnf*, while *disjunctive scheduling*, *packing* and *graceful graphs* clearly favour *gnf*. The opposite does not hold: there are no problems where *ref* could solve more instances than *gnf*.

Looking at the problems where the same number of instances were solved, the average solve time is much lower for *gnf* in *pattern matching*, *valve location* and *weighted sequence*, while being much higher in *no-mystery* and *sokoban*. A closer look at the raw data shows that for the latter benchmarks, the results are skewed by one instance which is solved orders of magnitude slower by *gnf*. A larger set of instances could show whether *gnf* is really significantly slower on these instances; a possible explanation is that crucial propagations are derived not soon enough. The average grounding size is

in line with the above results: problems with a much smaller grounding are typically solved much faster, and vice versa. Note that there are no benchmarks where using *gnf* leads to a larger average grounding size compared to *ref*. The above results imply that the described techniques are a significant improvement, with almost no instances negatively affected.

VI. RELATED WORK AND CONCLUSION

The presented work fits in a more general effort to combine techniques from SAT, CP and high-level knowledge representation languages. The solver-independent CP language Zinc [23] is grounded to the language MiniZinc[25], supported by a range of search algorithms using various paradigms, as can be seen on www.minizinc.org/challenge2012/results2012.html. In the context of CASP, several systems ground to ASP extended with constraint atoms, such as Clingcon[28] and EZ(CSP)[2]. For search, Clingcon combines the ASP solver Clasp[13] with the CSP solver Gecode[15], while EZ(CSP) combines an off-the-shelf ASP solver with an off-the-shelf CLP-Prolog system. The prototype CASP solver Inca[12] searches for answer sets of a ground CASP program by applying lazy clause generation for arithmetic and all-different constraints. As opposed to extending the search algorithm, a different approach is to transform a CASP program to a pure ASP program[11], afterwards applying any off-the-shelf ASP solver. CASP languages generally only allow a restricted set of expressions to occur in constraint atoms and impose conditions on where constraint atoms can occur. For example, none of the languages allows general atoms $P(\bar{c})$ with P an uninterpreted predicate symbol. One exception is the language $\mathcal{AC}(\mathcal{C})$ by [24], a language aimed at integrating ASP and Constraint Logic Programming. As shown in [19], the language captures the languages of both Clingcon and EZ(CSP); however, existing implementations only implement subsets of the language[16].

The presented ideas only improve performance when function symbols are present in the input theory. However, modelers are free to use predicates when some of its arguments depend functionally on each other and might choose to do so for example out of preference or ignorance of the functional relationship. In [6], it is investigated how functional relationships can be detected automatically, using a technique based on theorem proving, and how to subsequently rewrite the theory to introduce function symbols. Interesting avenues of future work are an experimental comparison with the above-mentioned systems and to investigate the effect of improving propagation for rules such as **Encode_{general}** and **Unfounded'**, which now only fire when most terms are assigned.

In this paper, we first presented a FO(\cdot) grounding algorithm, parametrized by the function symbols allowed in the grounding. In this way, we can, without changes to the input language, support the next generation of search algorithms that integrate techniques from SAT, ASP and CP. Second, we presented a search algorithm for the ground fragment of FO(\cdot). To the best of our knowledge, this is the first implementation for the full ground fragment of FO(\cdot) (combining definitions with nested uninterpreted functions), which is one of the first freely available implementations of lazy clause generation. Experimental results show that the grounding size can be significantly reduced while obtaining similar or improved search performance.

¹⁰Experiments were run on a 64-bit Ubuntu 12.04 system with an Intel Core i5 3570 processor and 8 GB of RAM. All experimental data is available at dtai.cs.kuleuven.be/krr/research/experiments

REFERENCES

- [1] K. R. Apt, *Principles of Constraint Programming*. Cambridge University Press, 2003.
- [2] M. Balduccini, “Industrial-size scheduling with asp+cp,” in *LPNMR*, 2011, pp. 284–296.
- [3] C. Baral, *Knowledge Representation, Reasoning, and Declarative Problem Solving*. New York, NY, USA: Cambridge University Press, 2003.
- [4] B. Bogaerts, B. De Cat, S. De Pooter, and M. Denecker, “IDP website,” <http://dtai.cs.kuleuven.be/krr/software/idp>, 2012.
- [5] F. Calimeri, G. Ianni, and F. Ricca, “The third open answer set programming competition,” *CoRR*, vol. abs/1206.3111, 2012.
- [6] B. De Cat and M. Bruynooghe, “Detection and exploitation of functional dependencies for model generation,” submitted to the 29th International Conference On Logic Programming.
- [7] B. De Cat and M. Mariën, “MINISAT(ID) website,” <http://dtai.cs.kuleuven.be/krr/software/minisatid>, 2008.
- [8] M. Denecker, Y. Lierler, M. Truszczynski, and J. Vennekens, “A tarskian informal semantics for asp,” in *Technical Communications of the 28th International Conference on Logic Programming*, 2012.
- [9] M. Denecker and E. Ternovska, “A logic of nonmonotone inductive definitions,” *ACM Transactions on Computational Logic (TOCL)*, vol. 9, no. 2, pp. 14:1–14:52, Apr. 2008.
- [10] M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczynski, “The second Answer Set Programming competition,” in *LPNMR*, 2009, pp. 637–654.
- [11] C. Drescher and T. Walsh, “Conflict-driven constraint answer set solving with lazy nogood generation,” in *AAAI*, 2011.
- [12] —, “Answer set solving with lazy nogood generation,” in *ICLP (Technical Communications)*, 2012, pp. 188–200.
- [13] M. Gebser, B. Kaufmann, and T. Schaub, “Conflict-driven answer set solving: From theory to practice,” *Artif. Intell.*, vol. 187, pp. 52–89, 2012.
- [14] M. Gebser, M. Ostrowski, and T. Schaub, “Constraint answer set solving,” in *ICLP*, ser. LNCS, P. M. Hill and D. S. Warren, Eds., vol. 5649. Springer, 2009, pp. 235–249.
- [15] Gecode Team, “Gecode: Generic constraint development environment,” 2013, available from <http://www.gecode.org>.
- [16] M. Gelfond, V. S. Mellarkod, and Y. Zhang, “Systems integrating answer set programming and constraint programming,” in *LaSh*, M. Denecker, Ed., 2008, pp. 145–152.
- [17] P. V. Hentenryck and J.-P. Carillon, “Generality versus specificity: An experience with ai and or techniques,” in *AAAI*, H. E. Shrobe, T. M. Mitchell, and R. G. Smith, Eds. AAAI Press / The MIT Press, 1988, pp. 660–664.
- [18] “The IDP system,” <http://dtai.cs.kuleuven.be/krr/software>, 2012.
- [19] Y. Lierler, “On the relation of constraint answer set programming languages and algorithms,” in *AAAI*, J. Hoffmann and B. Selman, Eds. AAAI Press, 2012.
- [20] G. Liu, T. Janhunen, and I. Niemelä, “Answer set programming via mixed integer programming,” in *KR*, G. Brewka, T. Eiter, and S. A. McIlraith, Eds. AAAI Press, 2012.
- [21] M. Mariën, “Model generation for ID-logic,” Ph.D. dissertation, Department of Computer Science, K.U.Leuven, Leuven, Belgium, February 2009.
- [22] M. Mariën, J. Wittocx, M. Denecker, and M. Bruynooghe, “SAT(ID): Satisfiability of propositional logic extended with inductive definitions,” in *SAT*, 2008, pp. 211–224.
- [23] K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. G. de la Banda, and M. Wallace, “The design of the Zinc modelling language,” *Constraints*, vol. 13, no. 3, pp. 229–267, 2008.
- [24] V. S. Mellarkod, M. Gelfond, and Y. Zhang, “Integrating answer set programming and constraint logic programming,” *Annals of Mathematics and Artificial Intelligence*, vol. 53, no. 1-4, pp. 251–287, 2008.
- [25] N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck, and G. Tack, “Minizinc: Towards a standard CP modelling language,” in *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, ser. LNCS, C. Bessiere, Ed., vol. 4741. Springer-Verlag, 2007, pp. 529–543.
- [26] I. Niemelä, “Answer set programming: A declarative approach to solving search problems,” in *JELIA*, 2006, pp. 15–18, invited talk.
- [27] M. Ostrowski and T. Schaub, “Asp modulo csp: The clingcon system,” *TPLP*, vol. 12, no. 4-5, pp. 485–503, 2012.
- [28] —, “Asp modulo csp: The clingcon system,” *TPLP*, vol. 12, no. 4-5, pp. 485–503, 2012.
- [29] N. Pelov, “Semantics of logic programs with aggregates,” Ph.D. dissertation, K.U.Leuven, Leuven, Belgium, April 2004.
- [30] P. J. Stuckey, “Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving,” in *CPAIOR*, 2010, pp. 5–9.
- [31] P. Vaezipoor, D. Mitchell, and M. Mariën, “Lifted unit propagation for effective grounding,” *CoRR*, vol. abs/1109.1317, 2011.
- [32] A. Van Gelder, “The alternating fixpoint of logic programs with negation,” *Journal of Computer and System Sciences*, vol. 47, no. 1, pp. 185–221, 1993.
- [33] A. Van Gelder, K. A. Ross, and J. S. Schlipf, “The well-founded semantics for general logic programs,” *Journal of the ACM*, vol. 38, no. 3, pp. 620–650, 1991.
- [34] J. Wittocx, M. Denecker, and M. Bruynooghe, “Constraint propagation for first-order logic and inductive definitions,” *ACM Transactions on Computational Logic*, 2013, accepted. [Online]. Available: <https://lirias.kuleuven.be/handle/123456789/369791>
- [35] J. Wittocx, M. Mariën, and M. Denecker, “Grounding FO and FO(ID) with bounds,” *Journal of Artificial Intelligence Research*, vol. 38, pp. 223–269, 2010.