2013 IEEE 25th International Conference on Tools with Artificial Intelligence

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 (MX) 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 $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, the knowledge representation language that extends First-Order Logic (FO) with, among others, inductive definitions, arithmetic and aggregates. An MX algorithm is developed, consisting of (i) a grounding algorithm for $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, parametrised by the function symbols allowed to occur in the reduced theory, and (ii) a search algorithm for unrestricted, ground $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$. The ideas are implemented in the IDP knowledge-base system and experimental evaluation shows that both more compact groundings and improved search performance are obtained.

Keywords*—*Model Expansion, Constraint Programming, Knowledge Representation, Grounding

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, known domain is extended into a total structure satisfying a given theory. These paradigms are studied in the fields of Constraint Programming (CP) [1], Answer Set Programming (ASP) [21] and Knowledge Representation (KR) [3].

A state-of-the-art approach is to reduce the input theory, formulated in an expressive logic, to a theory in a fragment of the language supported by some search algorithm while preserving a suitable form of equivalence. Afterwards, this algorithm searches for models of the theory. For example, model generation/expansion for the language $\mathrm{FO}(\cdot)$ [8] is performed by reducing theories to a ground fragment of $\mathrm{FO}(\cdot)$ 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 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 $pos_x(id, x)$ to express that the top-left corner of square id is at x and $sz(id)$ for the size of id:

$$\begin{aligned}&\forall id_1\ id_2\ x_1\ x_2 : id_1 \neq id_2 \land pos_x(id_1, x_1) \land pos_x(id_2, x_2)\\ &\quad\Rightarrow (x_1 + sz(id_1) \leq x_2 \lor x_2 + sz(id_2) \leq x_1).\end{aligned}$$

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

$$\begin{aligned}\forall id_1\ id_2 : id_1 \neq id_2 \Rightarrow (&f_x(id_1) + sz(id_1) \leq f_x(id_2)\\ &\lor f_x(id_2) + sz(id_2) \leq f_x(id_1)).\end{aligned}$$

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) [22]. In CASP, the ASP language is extended with *constraint atoms*, atoms that stand for the constraints of a CSP problem [15], [12], 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 [22], EZ(CSP) [2], Mingo [16] and Inca [9].

In this paper, we work in the context of the language $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, the language of the knowledge-base system (KBS) IDP [4]. $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ extends FO with, among others, inductive definitions, aggregates and arithmetic. We show that for $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, 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, $f_x(id_1) + sz(id_1) \leq f_x(id_2)$ is such an atom, for which efficient propagation techniques exist in the field of CP. We present a model expansion algorithm for $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ that exploits this idea. It consists of (i) an algorithm to ground

---

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

1082-3409/13 $31.00 © 2013 IEEE
DOI 10.1109/ICTAI.2013.159

$\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ theories without eliminating all function symbols from the grounding and (ii) a search algorithm for general, ground $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$. As different search algorithms often support different sets of function symbols, the grounding algorithm is *parametrised* 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) [17] using the technique of Lazy Clause Generation (LCG) [24], 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 its search algorithm MINISAT(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 ASP[1]. 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 $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ 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 $\mathrm{FO}(\cdot)$ [8] denotes the family of extensions of FO with new language constructs. The language we consider in this paper is the language $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$. It is a many-sorted extension of FO with aggregate functions, arithmetic and inductive definitions. We now 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 voc($\mathcal{T}$). Given two tuples $\overline{x}$ and $\overline{x}'$ of terms of equal length n, $\overline{x} = \overline{x}'$ is a shorthand for $\bigwedge_{i\in[1,n]} x_i = x'_i$. 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 $P(\overline{\tau})$ consists of two disjoint subsets of $D_{\tau_1} \times \cdots \times D_{\tau_n}$, denoted as P_{ct} and P_{cf}. An (partial) interpretation for a function symbol $f(\overline{\tau}) : \tau'$ is a function mapping elements of $D_{\tau_1} \times \cdots \times D_{\tau_n}$ 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 $P(\overline{d})$ is true in $\mathcal{I}$ if $P(\overline{d}) \in P^{\mathcal{I}}_{ct}$, false if $P(\overline{d}) \in P^{\mathcal{I}}_{cf}$ and unknown otherwise. An atom $f(\overline{d}) = d'$ is true in $\mathcal{I}$ if $\{d'\} = f^{\mathcal{I}}(\overline{d})$, unknown if $\{d'\} \subsetneq f^{\mathcal{I}}(\overline{d})$ 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. In this paper, we only consider interpretations where all types are finitely interpreted and totally ordered.

We extend the notion of term to include aggregate terms. A set expression is of the form $\{\overline{x} : \varphi : t\}$, $\{\varphi : t\}$ (if there are no local variables) or a union of set expressions $\{\overline{x}_1 : \varphi_1 : t_1\} \cup \{\overline{x}_2 : \varphi_2 : t_2\}$ (denoted shortly as $\{\overline{x}_1 : \varphi_1 : t_1, \overline{x}_2 : \varphi_2 : t_2\}$). Given an interpretation I and an assignment $\overline{d}$ to the free variables $\overline{y}$ of the set expression, the interpretation $\{\overline{x} : \varphi[\overline{y}/\overline{d}] : t[\overline{y}/\overline{d}]\}^I$ is the multiset $\{t[\overline{x}/\overline{d}', \overline{y}/\overline{d}]^{\mathcal{I}} \mid \varphi[\overline{x}/\overline{d}', \overline{y}/\overline{d}]^{\mathcal{I}} = \mathbf{t}\}$. Thus, in the context of a given assignment for the variables $\overline{y}$, the expression denotes the multiset of tuples t for which φ holds. Aggregate terms are of the form $agg(S)$, 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, nested aggregate terms occurring in a definition cannot contain any symbols defined in that definition.

Definitions Δ are sets of rules of the form $\forall \overline{x} : P(\overline{t}) \leftarrow \varphi$, where $P(\overline{t})$ 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 [23]. 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 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 $\forall \overline{x}_i : P(\overline{t}_i) \leftarrow \varphi_i$ with $i \in [1, n]$, is the set consisting of the sentence $\forall \overline{x}_i : \varphi_i \Rightarrow P(\overline{t}_i)$ for each $i \in [1, n]$ and the

---

[1] A formal comparison of $\mathrm{FO}(\cdot)$ and ASP is presented in [7]

sentence $\forall \overline{x} : P(\overline{x}) \Rightarrow \bigvee_{i \in [1,n]} (\overline{x} = \overline{t}_i \wedge \varphi_i)$. It is well-known that Δ entails the completion of all its defined symbols, 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 implications and equivalences are eliminated, $\neg$ only occurs directly in front of atoms and conjunctions and disjunctions are in left-associative form. We assume, without loss of generality, our sentences and rule bodies are in 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 initialised 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. $\neg\neg\varphi \rightarrowtail \varphi$ 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.

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

These rewrite rules ensure that all occurrences of function symbols $f \notin$ ResF are top left symbols in equalities $f(\overline{t}) = t$. Note, if ResF is empty, such atoms are of the form $f(\overline{t}) = t$ with $t_1, \ldots, t_n, t$ 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 $f/n \notin$ ResF, we introduce a new predicate symbol $g_f/n+1$, apply the rewrite rule $f(\overline{t}) = t \rightarrowtail g_f(\overline{t}, t)$, add $\forall \overline{x} : \#(\{y : g_f(\overline{x}, y) : 1\}) = 1$ to $\mathcal{T}_g$ and add $\forall \overline{x}\ y : f(\overline{x}) = y \Leftrightarrow g_f(\overline{x}, y)$ to $\mathcal{T}_m$.

B. Phase 2: Grounding

From now on, the domains of variables are made explicit in all expressions, written as $\forall \overline{x} \in \overline{D} : \varphi$ or $\{\overline{x} \in \overline{D} : \varphi : t\}$. Initially, $\overline{D}$ is $\tau_1^{\mathcal{I}_{in}} \times \cdots \times \tau_n^{\mathcal{I}_{in}}$, where τ_i is the type of x_i[2].

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

- **Split** *conjunctive sentences:* $\varphi \wedge \psi \rightarrowtail \varphi, \psi$ where $\varphi \wedge \psi$ is a sentence in $\mathcal{T}_g$.
- **Instantiate**, for some $\overline{d} \in \overline{D}$:
  $\forall \overline{x} \in \overline{D} : \psi \rightarrowtail \psi[\overline{x}/\overline{d}] \wedge \forall \overline{x} \in \overline{D} - \overline{d} : \psi$.
  $\exists \overline{x} \in \overline{D} : \psi \rightarrowtail \psi[\overline{x}/\overline{d}] \vee \exists \overline{x} \in \overline{D} - \overline{d} : \psi$.
  $\forall \overline{x} \in \overline{D} : A \leftarrow \psi \rightarrowtail A[\overline{x}/\overline{d}] \leftarrow \psi[\overline{x}/\overline{d}]$
  $\qquad \forall \overline{x} \in \overline{D} - \overline{d} : A \leftarrow \psi$.
  $\{\overline{x} \in \overline{D} : \varphi : t\} \rightarrowtail \{\varphi[\overline{x}/\overline{d}] : t[\overline{x}/\overline{d}]\} \cup$
  $\qquad \{\overline{x} \in \overline{D} - \overline{d} : \varphi : t\}$.
- **Simplify**

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

- **Introduce Tseitin** $\psi \rightarrowtail T_\psi$, 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** $t \rightarrowtail c_t$, where t is an occurrence of a term without free variables in $\mathcal{T}_g$ and c_t is a newly 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 $P(\overline{e})$, $f(\overline{e}) \sim e_0$ or $agg(\{L_1 : e_1\} \cup \cdots \cup \{L_n : e_n\}) \sim e_0$.
- **Evaluate** $t \rightarrowtail t^{\mathcal{I}_{in}}$ if t is a domain term with a single value in $t^{\mathcal{I}_{in}}$. $P(\overline{d}) \rightarrowtail P(\overline{d})^{\mathcal{I}_{in}}$, if $P(\overline{d})^{\mathcal{I}_{in}} \neq \mathbf{u}$. The rule is not applied to $P(\overline{d})$ in definitions defining P.[3]

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

---

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

[3] By definition, **Evaluate** checks well-typedness of expressions.

following forms (with all L_i's domain literals):

$$L_1 \vee \ldots \vee L_n. \quad Q(\overline{e}). \quad f(\overline{e}) \sim e_0.$$
$$agg(\{L_1 : e_1\} \cup \cdots \cup \{L_n : e_n\}) \sim e_0.$$
$$P(\overline{e}) \leftarrow L_1 \wedge \ldots \wedge L_n.$$
$$P(\overline{e}) \leftarrow L_1 \vee \ldots \vee L_n.$$
$$P(\overline{e}) \leftarrow Q(\overline{e}'). \qquad P(\overline{e}) \leftarrow f(\overline{e}) \sim e_0.$$
$$P(\overline{e}) \leftarrow agg(\{L_1 : e_1\} \cup \cdots \cup \{L_n : e_n\}) \sim e_0.$$

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 and that each application results in a theory ordered strictly lower. 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 minimises 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 minimise grounding size, followed by minimising running time and memory usage.
- **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 minimised. E.g., by avoiding creating different Tseitin symbols for different occurrences of the same formula.

A useful 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 $\mathcal{T}_{in}$ [27]. With the refined structure $\mathcal{I}_{in}$', the ground theories are sometimes orders of magnitude smaller than w.r.t. $\mathcal{I}_{in}$ [28].

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 $\mathcal{I}_{in}$.[4] The algorithm is based on existing MX algorithm for function-free GNF, implemented in the system MINISAT(ID), described in [17]. That algorithm is a Conflict-Driven Clause-Learning (CDCL) search algorithm, extended to handle inductive definitions and aggregates. Recall that running the algorithm in the previous section with $\mathrm{ResF} = \emptyset$ results in function-free GNF. As before, we assume that all types τ are interpreted as finite sets $\tau^{\mathcal{I}_{in}}$ of domain elements.

A. Adapting CDCL

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 $Q(\overline{e})$, $f(\overline{e}) \sim e_0$ or $agg(\{L_1 : e_1\} \cup \cdots \cup \{L_n : e_n\}) \sim e_0$, is generalised as an equivalence $P(\overline{d}) \Leftrightarrow A$. 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 $\Delta_1, \ldots, \Delta_n$ as shown in [8]. 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 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 [8]. **Unfounded** searches for unfounded sets [26] in a definition Δ and if an unfounded set U is found, propagates all its

---

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

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 [26]. 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 (LCG)*, presented in [24]. LCG 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 lazily generating GNF *sentences*.

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 $\forall \overline{x} : \varphi \Rightarrow L$; 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 $\overline{x}$ for which φ holds in $\mathcal{I}$ and $L^{\mathcal{I}}$ is not true. The algorithm then consists of adding the sentence $\varphi[\overline{x}/\overline{d}] \Rightarrow L[\overline{x}/\overline{d}]$ for the relevant instantiations $\overline{d}$ of $\overline{x}$. 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 *order encoding* [25]. A domain term c of the form $f(\overline{d})$ with f mapping to the domain $D = \{d_1, \ldots, d_n\}$ is encoded by the predicate symbol $T_{c\leq}/1$. Trivially, any atom $T_{c\leq}(d)$ with $d \notin D$ is interpreted true if $d > d_n$, false if $d < d_1$ and as $T_{c\leq}(d')$ otherwise, with d' the domain element in D closest to d but still smaller. In the sequel, we use $\lceil c \leq t \rceil$, with t a term, to denote the atom $T_{c\leq}(t)$. All other comparison operators $\sim$ can be defined in terms of $\leq$. We use $\lceil c \sim d \rceil$ as a shorthand for those rewritings. E.g, $\lceil c \neq d_i \rceil$ denotes $\lceil c \leq d_{i-1} \rceil \vee \neg \lceil c \leq d_i \rceil$. The dependencies between different atoms is expressed by the following sentences:

$$\begin{array}{l} \forall x \in D - d_n : \lceil c \leq x \rceil \Rightarrow \lceil c \leq \mathit{next}(x) \rceil . \\ \forall x \in D - d_1 : \lceil c > x \rceil \Rightarrow \lceil c > \mathit{prev}(x) \rceil . \end{array}$$

The propagation rule **Encode** is applied to a domain term c 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 [24]. 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 $\forall x \in D : \lceil f(\overline{d}) = x \rceil \Rightarrow f(\overline{d}) = x$ to $\mathcal{T}_m$.

For each c, we define $\min_c$ and $\max_c$ as $\max\{d \in D \mid \lceil c \geq d \rceil^{\mathcal{I}} = \mathbf{t}\}$, respectively $\min\{d \in D \mid \lceil c \leq d \rceil^{\mathcal{I}} = \mathbf{t}\}$. The range of c is then defined as $[\min_c, \max_c]$. The values $\min_c$ and $\max_c$ can be computed from $\mathcal{I}$, but an efficient algorithm should store them and adapt them incrementally whenever $\mathcal{I}$ changes.

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 $\mathcal{T}_g$.

The order encoding is selected over encoding the function as a set of equalities $T_{c=}(d_i)$ as the encoding of inequalities is more compact 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 [24].

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.

$$\begin{array}{lll} \forall x \in D \cup D' : & \lceil c \leq x \rceil \wedge \lceil c' \geq x \rceil & \Rightarrow P. \\ \forall x \in D \cup D' : & \lceil c > x \rceil \wedge \lceil c' < x \rceil & \Rightarrow \neg P. \\ \forall x \in D : & \lceil c' \leq x \rceil \wedge P & \Rightarrow \lceil c \leq x \rceil . \\ \forall x \in D : & \lceil c' \geq x \rceil \wedge \neg P & \Rightarrow \lceil c > x \rceil . \\ \forall x \in D' : & \lceil c \geq x \rceil \wedge P & \Rightarrow \lceil c' \geq x \rceil . \\ \forall x \in D' : & \lceil c \leq x \rceil \wedge \neg P & \Rightarrow \lceil c' < x \rceil . \end{array}$$

It is easy to see that together with the encoding of c and c', this set of sentences it $\{\Sigma, \mathcal{I}\}$-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. Rule **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 $\lceil c \geq 8 \rceil$ to $\mathcal{I}$', **Comparison** checks for $x = 8$ which of the left-hand sides are true, which is the case for the sentence $\lceil c \geq 8 \rceil \wedge P \Rightarrow \lceil c' \geq 8 \rceil$. As P is true in $\mathcal{I}$', the sentence is added to $\mathcal{T}_s$ and **UP** derives $\lceil c' \geq 8 \rceil$.

E. Aggregates

Next, we introduce propagation rules for sentences of the form $P \Leftrightarrow agg(\{L_1 : e_1\} \cup \cdots \cup \{L_n : e_n\}) \leq e_0$ 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$_{\text{max}}$** rewrites a maximum constraint $P \Leftrightarrow \max(S) \leq c$ into the following sentences

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

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$_{\text{sum}}$** propagation rule, is out of the scope of this paper, we only give an example. For some set $S_{in} \subseteq [1, n]$, with S_c its complement, the sentence

$$\forall \overline{x} : \left(\bigwedge_{i \in S_{in}} L_i \wedge \lceil e_i \leq x_i \rceil \right) \wedge \left\lceil e_0 \geq \sum_{i \in S_{in}} x_i \right\rceil \bigwedge_{i \in S_c} \neg L_i \Rightarrow P$$

expresses that P has to be true if e_0 is larger or equal than the sum of the maxima of all terms with a true condition (indices S_{in}) and all other conditions are false. 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 $P \Leftrightarrow q(\overline{e})$ and $P \Leftrightarrow f(\overline{e}) \sim e_0$ are handled by waiting until all domain terms in $\overline{e}$ are assigned. At that moment, the instantiated constraint is generated, which coincides with instantiations of the sentence[5]

$$\begin{array}{l} \forall \overline{x} \in dom_{\overline{e}} : \lceil \overline{e} = \overline{x} \rceil \Rightarrow (P \Leftrightarrow Q(\overline{x})), \text{respectively} \\ \forall \overline{x} \in dom_{\overline{e}} : \lceil \overline{e} = \overline{x} \rceil \Rightarrow (P \Leftrightarrow f(\overline{x}) \sim e_0). \end{array}$$

The propagation rule **Encode$_{\text{general}}$** adds the above sentences whenever the value of each of the c_i is known (applying Tseitin introduction to generate sentences in GNF).

Example IV.3. The *element constraint* `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.

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 $\{P(\overline{e}_1) \leftarrow \varphi_1, \ldots, P(\overline{e}_n) \leftarrow \varphi_n\}$. The completion of P for Δ is then the (non-ground) sentence $\forall \overline{x} : P(\overline{x}) \Leftrightarrow \bigvee_{i \in [1,n]} \lceil \overline{e}_i = \overline{x} \rceil \wedge \varphi_i$. The rule **Completion′** adds the equivalent sentences

$$\begin{array}{l} \bigwedge_{i \in [1,n]} \varphi_i \Rightarrow P(\overline{e}_i) \\ \forall \overline{x} : P(\overline{x}) \Rightarrow \left(\bigvee_{i \in [1,n]} \lceil \overline{e}_i = \overline{x} \rceil \wedge \varphi_i \right) \end{array}$$

The former sentence is added eagerly for each i (as it is already ground). For the latter sentence, **Completion′** adds its instantiation of $\overline{x}$ with $\overline{d}$ to $\mathcal{T}_s$ for atoms $P(\overline{d})$ true in $\mathcal{I}$.

An issue with the condition on instantiation is that propagations might be missed. Indeed, the latter **Completion′** sentence is only instantiated for $P(\overline{d})$ true in $\mathcal{I}$; however, if $\left(\bigvee_{i \in [1,n]} \lceil \overline{e}_i = \overline{d} \rceil \wedge \varphi_i \right)$ is false, then $\neg P(\overline{d})$ is entailed. If $P(\overline{d})$ 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 soundness.

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 EC, this clause is only valid conditionally, as we had to substitute several constants in order to obtain it. So instead of adding EC to $\mathcal{T}_s$, we add

$$\bigwedge_{c | c \text{ occurs in } \Delta} \lceil c \neq c^{\mathcal{I}} \rceil \Rightarrow EC.$$

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, all nodes reachable from the start node through next, defined[6] as $\{r(\mathrm{start}). \quad \forall x : r(\mathrm{next}(x)) \leftarrow r(x).\}$. In the context of an interpretation $\mathcal{I}$ over domain $\{a, b, c\}$, with $\mathrm{start}^{\mathcal{I}} = a$ and $\mathrm{next}^{\mathcal{I}} = \{a \mapsto b, b \mapsto a, c \mapsto c\}$, the definition reduces to the rules $r(a)$, $r(b) \leftarrow r(a)$, $r(a) \leftarrow r(b)$, and $r(c) \leftarrow r(c)$, to which **Unfounded** can be applied. As **Unfounded** would derive $\neg r(c)$, **Unfounded′** generates $(\lceil \mathrm{start} = a \rceil \wedge \lceil \mathrm{next}(a) = b \rceil \wedge \lceil \mathrm{next}(b) = a \rceil \wedge \lceil \mathrm{next}(c) = c \rceil) \Rightarrow \neg r(c)$.

H. Pre-interpretation over some symbols

As discussed above, the grounding algorithm gets as input a partial, consistent input interpretation $\mathcal{I}_{in}$, parts of which

---

[5] We use $\lceil \overline{e} = \overline{x} \rceil$ as a shorthand for $\bigwedge_{i \in [1, |\overline{e}|]} \lceil e_i = x_i \rceil$.

[6] Note 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.

are implicit (e.g. interpretations of numerical functions) or described symbolically (e.g. ranges $0..n$). The information in $\mathcal{I}_{in}$ should be passed to the solver, but we do not want to add $\mathcal{I}_{in}$ as constraints to the theory, for the same reason as we do not want to eagerly generate the full propositional grounding. Instead, the following propagation rule takes care of adding just enough of $\mathcal{I}_{in}$ to obtain interpretations 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 $P(d)$ or $\neg P(d)$ for every domain-element $d \in D$ would cause an immense ground theory. 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 propagation rules, a search algorithm consists of an execution order $\ll$ on those rules. The execution order can have a great impact on the efficiency of the search. E.g., whenever **Fail** is possible, it is useless to propagate further; **UP** is preferred over **Unfounded** because it is 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 such that propagation on simpler constraints is performed before propagation on more complex constraints, resulting in the following order (recall, **Encode_{max}** is executed in the initial phase).

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

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 $\mathcal{I}_{in}$.*

V. Experiments

The grounding algorithm is implemented in IDP^3 [4], a knowledge-base system supporting state-of-the-art model expansion, as can be observed from e.g. the previous ASP competition [5][8]. The search algorithm is implemented as the newest version of the MiniSAT(ID) solver, originally described in [17]. As benchmarks, we used the benchmarks and instances of the fourth ASP competition[9] in the NP complexity class, the classic CP benchmarks disjunctive scheduling and square-packing of the third ASP competition and a (new) concrete-delivery scheduling application. The search implementation is currently limited to functions over integer domains, a constraint satisfied in all considered benchmarks.

Each of the benchmarks and associated instances was solved using IDP^3, measuring performance and size of the grounding[10]. Two different setups were used. The `ground` setup applies MX with ResF the set of all functions with integer domains. The `ground` setup, able to apply all ideas presented in this paper, is compared to the (function-free) reference setup `prop`, which uses $\text{ResF} = \emptyset$, resulting in a (pseudo-) propositional grounding. As discussed earlier, this results in an effectively propositional grounding, in which the search collapses to the original MiniSAT(ID) algorithm.

Benchmark	# inst.	# solved	avg. time(sec)	avg. size (# atoms)
bottle fill.	30	30(30)	99(98)	$9 \times^{5} (1 \times 10^{6})$
graceful graphs	30	**19**(3)	**131**(489)	$\mathbf{8 \times 10^{5}} (3 \times 10^{7})$
incr. sched.	30	**20**(8)	3(1)	$\mathbf{5 \times 10^{3}} (- - -)$
no-mystery	30	27(28)	52(64)	$2 \times 10^{5} (3 \times 10^{5})$
pattern matching	30	**30**(22)	**3**(44)	$\mathbf{5 \times 10^{4}} (1 \times 10^{7})$
ricochet robots	30	15(15)	402(408)	$2 \times 10^{7} (2 \times 10^{7})$
sokoban	30	17(17)	115(113)	$\mathbf{5 \times 10^{5}} (5 \times 10^{5})$
solitaire	27	23(22)	10(9)	$3 \times 10^{4} (3 \times 10^{4})$
stable marriage	30	30(30)	124(123)	$3 \times 10^{7} (3 \times 10^{7})$
weighted seq.	30	30(30)	3(12)	$\mathbf{2 \times 10^{3}} (5 \times 10^{5})$
disj. scheduling	21	**21**(5)	**2**(27)	$\mathbf{3 \times 10^{3}} (1 \times 10^{7})$
packing	30	**30**(9)	**1**(138)	$\mathbf{1 \times 10^{4}} (1 \times 10^{7})$
crossing min.*	30	9(11)	**48**(128)	$\mathbf{8 \times 10^{3}} (4 \times 10^{5})$
still life*	26	3(4)	**1**(41)	$\mathbf{1 \times 10^{4}} (5 \times 10^{4})$
valve location*	30	**6**(2)	156(**49**)	$\mathbf{2 \times 10^{6}} (6 \times 10^{6})$
concrete deliv.*	30	**18**(0)	**171**(−−)	$\mathbf{7 \times 10^{5}} (- - -)$

Table I. Experimental results, formatted as ground(prop), with clear winners in bold. For optimization problems (*), # solved and avg. time reflect solved to optimality.

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. The time limit was 1000 seconds, the memory limit 5 GB.

The most important conclusion this table gives, is that the techniques described in this paper are crucial to solve some problems, such as *graceful graphs* and *concrete delivery*, while *disjunctive scheduling*, *packing* and *incremental scheduling* clearly favour `ground`. The opposite does not hold: there are no problems where `prop` could solve significantly more instances than `ground`. Looking at the problems where the same number of instances were solved, in all benchmarks the average solve time is similar or often significantly better for `ground`. 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 `ground` leads to a larger average grounding size compared to `prop`.

For the optimization problems, it is also interesting to compare the best solutions found by both approaches within the time limit, even if the optimal solution was not found (or proven to be optimal). For *still-life*, the best solution was found by `prop` in 18 cases opposed to only once by `ground`, and similarly for *crossing-minimisation* (14 to 4). The situation is reversed for *valve location* (3 to 27) and *concrete delivery* (0 to 26), where `ground` clearly outperforms `prop`. The reason is that the rules in `ground` detect propagation later in some cases, causing more suboptimal models to be found.

The above results imply that the described techniques are a

---

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

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

[9] Available at https://www.mat.unical.it/aspcomp2013/OfficialProblemSuite

[10] Experiments 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

significant improvement in almost all cases, although the naive propagation rules should be improved further.

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 [18] is grounded to the language MiniZinc [20], 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 [22] and EZ(CSP) [2]. For search, Clingcon combines the ASP solver Clasp [11] with the CSP solver Gecode [13], while EZ(CSP) combines an off-the-shelf ASP solver with an off-the-shelf CLP-Prolog system. The prototype CASP solver Inca [9] searches for answer sets of a ground CASP program by applying LCG 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 [10], 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 $P(\overline{c})$ with P an uninterpreted predicate symbol. One exception is $\mathcal{AC}(\mathcal{C})$, a language aimed at integrating ASP and Constraint Logic Programming [19]. As shown in [15], the language captures the languages of both Clingcon and EZ(CSP); however, only subsets of the language are implemented [14].

The presented ideas only improve performance when function symbols are present in the input theory. However, modellers are free to use predicates when some of its arguments depend functionally on each other and might choose to do so. 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 topics for future work are an experimental comparison with the above-mentioned systems and an investigation of the effect of improved propagation for rules such as $\mathbf{Encode}_{\mathbf{general}}$ and $\mathbf{Unfounded}'$, which now only fire when all relevant terms are assigned.

To conclude, this paper first presented an FO($\cdot$) grounding algorithm, parametrised 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) and one of the first freely available implementations of LCG. Experimental results show that the grounding size can be significantly reduced while obtaining similar or improved search performance.

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," accepted for the 29th International Conference On Logic Programming.

[7] M. Denecker, Y. Lierler, M. Truszczynski, and J. Vennekens, "A tarskian informal semantics for asp," in *International Conference on Logic Programming (Technical Communications)*, 2012.

[8] 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.

[9] C. Drescher and T. Walsh, "Conflict-driven constraint answer set solving with lazy nogood generation," in *AAAI*, 2011.

[10] ——, "Translation-based constraint answer set solving," in *IJCAI*, 2011, pp. 2596–2601.

[11] M. Gebser, B. Kaufmann, and T. Schaub, "Conflict-driven answer set solving: From theory to practice," *Artif. Intell.*, vol. 187, pp. 52–89, 2012.

[12] 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.

[13] Gecode Team, "Gecode: Generic constraint development environment," 2013, available from `http://www.gecode.org`.

[14] 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.

[15] Y. Lierler, "On the relation of constraint answer set programming languages and algorithms," in *AAAI*, J. Hoffmann and B. Selman, Eds. AAAI Press, 2012.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck, and G. Tack, "Minizinc: Towards a standard CP modelling language," in *CP'07*, ser. LNCS, C. Bessiere, Ed., vol. 4741. Springer, 2007, pp. 529–543.

[21] I. Niemelä, "Answer set programming: A declarative approach to solving search problems," in *JELIA*, 2006, pp. 15–18, invited talk.

[22] M. Ostrowski and T. Schaub, "Asp modulo csp: The clingcon system," *TPLP*, vol. 12, no. 4-5, pp. 485–503, 2012.

[23] N. Pelov, "Semantics of logic programs with aggregates," Ph.D. dissertation, K.U.Leuven, Leuven, Belgium, April 2004.

[24] P. J. Stuckey, "Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving," in *CPAIOR*, 2010, pp. 5–9.

[25] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara, "Compiling finite linear csp into sat," *Constraints*, vol. 14, no. 2, pp. 254–272, 2009.

[26] 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.

[27] J. Wittocx, M. Denecker, and M. Bruynooghe, "Constraint propagation for first-order logic and inductive definitions," *ACM Trans. Comput. Logic*, vol. 14, no. 3, pp. 17:1–17:45, Aug. 2013.

[28] 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.