GIDL: A Grounder for FO$^+$

Johan Wittocx[*] and **Maarten Mariën** and **Marc Denecker**
Department of Computer Science, K.U. Leuven, Belgium
{johan,maartenm,marcd}@cs.kuleuven.be

Abstract

In this paper, we present GIDL, a grounder for FO$^+$. FO$^+$ is a very expressive extension of first-order logic with several constructs such as inductive definitions, aggregates and arithmetic. We describe the input and output language of GIDL, and provide details about its architecture. In particular, the core grounding algorithm implemented in GIDL is presented. We compare GIDL with other FO$^+$ grounders and with grounders for Answer Set Programming.

Introduction and Motivation

The ambition of *declarative problem solving* is, in a nutshell, that a human expert represents his knowledge as a precise logic specification in terms of a vocabulary formalizing relevant objects and concepts of the problem domain, and solves computational tasks within this domain by applying suitable forms of logical inference on the logic specification. The success of a declarative problem solving framework depends on three main factors: the quality of the logic as a specification language, the flexibility of the logical inference to solve a broad class of computational problems, and the availability of efficient solvers.

An important and flexible logical inference task is finite (Herbrand) model generation. Indeed, in many real-life computational problems, one searches for objects of a complex nature, e.g., plans, schedules, assignments, etc. Such objects are often represented as (finite) structures. Model generation serves to explicitly construct such a structure, given an implicit description of it by means of a logic theory. The idea of a declarative problem solving framework based on computing "solutions" as the models of a theory was presented for the first time in (Marek & Truszczyński 1998) in the context of Answer Set Programming (ASP). Earlier, SAT-solvers had been used in this spirit, for example in Kautz and Selman's blackbox approach to planning problems (Kautz & Selman 1996). And, as recently pointed out in (Mitchell & Ternovska 2008), problem solving in Constraint Programming (CP) systems often amounts to computing models of first-order logic (FO) specifications.

In (Mitchell & Ternovska 2005), a declarative framework based on *model expansion* (MX) was presented. MX for a logic $\mathcal{L}$, denoted MX($\mathcal{L}$), extends model generation: it takes as input not only an $\mathcal{L}$-theory T over a vocabulary Σ and finite domain D, but also a structure I_σ with domain D, interpreting a subvocabulary $\sigma \subseteq \Sigma$. It searches to expand I_σ into a Σ-model of T. The input interpretation I_σ presents a convenient way to store *data* of a problem.

From a computational point of view, an interesting aspect of finite model generation and MX is that its complexity remains in NP for every logic for which the model checking problem is in P. This is the case for, e.g., first-order logic (FO) and many extensions of it, which are languages par excellence for describing many real-life computational problems. In this paper, we consider MX for such a logic, namely full first-order logic extended with aggregates, inductive definitions, arithmetic, partial functions and ordered sorts. We denote this logic by FO$^+$. Clearly, FO$^+$ is an expressive language, convenient for modelling a broad class of domains.

An important result in (Mitchell & Ternovska 2005) states that in the context of MX, FO is sufficient to solve all problems in NP. More precisely, for every NP decision problem on finite σ-structures, there exists a vocabulary $\Sigma \supseteq \sigma$ and a theory T over Σ such that a σ-structure I_σ is accepted iff there exists a model of T expanding I_σ. Hence, the class of problems that can be *represented* in MX(FO$^+$) and MX(FO) is exactly the same. In practice however, new language primitives, such as the ones in FO$^+$, may seriously ease the modelling task and enlarge the class of problems that can be *solved* by practical implementations. As an example, consider the concept of *reachability* in a graph, which is often needed to model, e.g., planning or scheduling problems. This concept can be expressed in MX(FO), but not in a simple and natural manner: it requires a non-trivial encoding of an iterative fixpoint construction in FO. To allow for a direct, natural representation, one can consider MX for FO(ID), an extension of FO with *inductive definitions* (Denecker 2000). Besides making the modelling task easier, the resulting MX problem can be solved more efficiently, at least by the current generation of solvers. A similar argument applies for other language primitives, such as aggregates and arithmetic.

Currently, most model generation systems, and hence also MX solvers, consist of two components: a *grounder* and a *propositional solver*. The grounder transforms the input to an equivalent propositional theory, whose models are then

[*]Research assistant of the *Fonds voor Wetenschappelijk Onderzoek - Vlaanderen* (FWO Vlaanderen)

computed by the propositional solver. Several grounders for (fragments of) MX(FO$^+$) are being developed. MX-IDL (Mariën, Wittocx, & Denecker 2006), the first implemented MX(FO(ID)) grounder, works by translating its input into an equivalent normal logic program, according to the transformation described in (Mariën, Gilis, & Denecker 2004), and then calls a (slightly adapted) grounder for ASP. MXIDL can handle full many-sorted FO(ID), extended with arithmetic. The first native grounding algorithm for MX(FO(ID)) was described in (Patterson *et al.* 2007), and partially implemented in the MXG system (Mitchell *et al.* 2006). MXG allows function-free FO, cardinality aggregates and a very restricted form of inductive definitions as input.

In this paper, we present GIDL, a new MX grounder, designed to handle a very expressive input language. It is tightly coupled with the propositional solvers MIDL (Mariën, Wittocx, & Denecker 2007) and MINISAT(ID) (Mariën *et al.* 2008), developed in our group. GIDL's input language is full FO$^+$: full order-sorted FO(ID), extended with cardinality, sum and product aggregates, partial functions and arithmetic. We present this input language in detail and describe GIDL's architecture. In particular, we present the core grounding algorithm, which is different from the one in MXG. We compare GIDL to MX-IDL and MXG, showing that it is currently the fastest MX grounder. We also compare GIDL to grounders for ASP and to PSGRND (East *et al.* 2006), a grounder for the logic of propositional schemata (East & Truszczynski 2006a).

Preliminaries

In this section, we present many-sorted FO and FO(ID) and formally define the concepts of model expansion and grounding. We assume the reader is familiar with standard FO.

Many-Sorted First-Order Logic with Equality

A *vocabulary* Σ consists of a set Σ_S of sorts, and of variables, constant, predicate and function symbols. Variables and constant symbols are denoted by lowercase letters, predicate and function symbols by uppercase letters. Sets and tuples of variables are denoted by $\overline{x}, \overline{y}, \ldots$. Each variable x and constant symbol c has an associated sort $\mathfrak{s}(x)$, respectively $\mathfrak{s}(c) \in \Sigma_S$, each predicate symbol P with arity n an associated tuple of sorts $\mathfrak{s}(P) \in \Sigma_S^n$, and each function symbol F with arity n an associated tuple $\mathfrak{s}(F) \in \Sigma_S^{n+1}$.

A *term* over a vocabulary Σ is inductively defined as follows:

- A variable x of Σ is a term of sort $\mathfrak{s}(x)$.
- A constant c of Σ is a term of sort $\mathfrak{s}(c)$.
- If F is a function symbol of Σ with $\mathfrak{s}(F) = (s_1, \ldots, s_n, s_{n+1})$, and $t_1, \ldots, t_n$ are terms over Σ of sort respectively $s_1, \ldots, s_n$, then $F(t_1, \ldots, t_n)$ is a term of sort s_{n+1}.

The sort of a term t is denoted by $\mathfrak{s}(t)$. A (well-sorted) FO formula over Σ is inductively defined by:

- If P is a predicate symbol with $\mathfrak{s}(P) = (s_1, \ldots, s_n)$ and $t_1, \ldots, t_n$ are terms of sort respectively $s_1, \ldots, s_n$, then $P(t_1, \ldots, t_1)$ is a formula.
- If t_1 and t_2 are two terms of the same sort, then $t_1 = t_2$ is a formula.
- If φ and ψ are formulas and x is a variable, then $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\exists x\ \varphi$ and $\forall x\ \varphi$ are formulas.

An *atom* is a formula of the form $P(\bar{t})$ or $t_1 = t_2$. A *literal* is an atom or the negation of an atom. An occurrence of a formula φ as subformula in a formula ψ is *positive* (*negative*) if it occurs in the scope of an even (odd) number of negations. For a formula φ, we often write $\varphi[\overline{x}]$ to indicate that $\overline{x}$ are its free variables. The formula $\varphi[x/c]$ is the formula obtained by replacing in φ all free occurrences of the variable x by the constant symbol c. This notation is extended to tuples of variables and constant symbols of the same length. A *sentence* is a formula without free variables.

A Σ-*interpretation* I consists of

- a domain s^I for each sort $s \in \Sigma$;
- a domain element $x^I \in s^I$ for each variable x with $\mathfrak{s}(x) = s$;
- a domain element $c^I \in s^I$ for each constant c with $\mathfrak{s}(c) = s$;
- a relation $P^I \in s_1^I \times \ldots \times s_n^I$ for each predicate symbol P with $\mathfrak{s}(P) = (s_1, \ldots, s_n)$;
- a function $F^I : s_1^I \times \ldots \times s_n^I \to s_{n+1}^I$ for each function symbol F with $\mathfrak{s}(F) = (s_1, \ldots, s_{n+1})$.

A Σ-*structure* is an interpretation of only the sorts, constant, relation and function symbols of Σ. The restriction of a Σ-interpretation I to a vocabulary $\sigma \subseteq \Sigma$ is denoted by $I|_\sigma$. For a variable x and domain element d, $I[x/d]$ is the interpretation that assigns d to x and corresponds to I on all other symbols. This notation is extended to variables and domain elements of the same length.

The value t^I of a term t in an interpretation I, and the satisfaction relation $\models$ are defined as usual (see, e.g., (Enderton 1972)).

Inductive Definitions and FO(ID)

FO(ID) (Denecker 2000) is an extension of FO with inductive definitions. It can be viewed as an integration of FO with logic programming.

A *definition* over a vocabulary Σ is a finite set of rules of the form

$$\forall \overline{x}\ (P(\bar{t}) \leftarrow \varphi[\overline{y}]),$$

where φ is an FO formula over Σ, $\overline{y} \subseteq \overline{x}$, P is a predicate in Σ and $\bar{t}$ a tuple of terms such that $P(\bar{t})$ is well-sorted. Also, the set of variables occurring in $\bar{t}$ is a subset of $\overline{x}$. $P(\bar{t})$ is called the *head* of the rule, φ the *body*. The connective $\leftarrow$ is called *definitional implication* and is to be distinguished from material implication $\supset$. A predicate appearing in the head of a rule of a definition Δ is called a *defined* predicate of Δ, any other predicate symbol and each constant and function symbol is called an *open symbol* of Δ. The set of open symbols of Δ is denoted by $\mathrm{Open}(\Delta)$, the set of defined predicates by $\mathrm{Def}(\Delta)$. An occurrence of a formula φ

in a rule body is positive (negative) if it occurs in the scope of an odd (even) number of negations.

A Σ-interpretation I is said to satisfy a definition Δ over Σ, denoted $I \models \Delta$, if $I|_{\mathrm{Def}(\Delta)}$ is the well-founded model of Δ extending $I|_{\mathrm{Open}(\Delta)}$. The definition of well-founded model can be found in (Van Gelder, Ross, & Schlipf 1991; Denecker & Ternovska 2004).

An FO(ID) theory T is a finite set of FO sentences and definitions. An interpretation is a model of T iff it satisfies all sentences and definitions of T.

Model Expansion

Model expansion for a logic $\mathcal{L}$, abbreviated MX($\mathcal{L}$), was first presented as a declarative problem solving paradigm in (Mitchell & Ternovska 2005). For representation theorems, like the *capturing NP* property mentioned in the introduction, and for a comparison with other paradigms, we refer the reader to that paper.

Definition 1 (MX($\mathcal{L}$)). Given an $\mathcal{L}$ theory over a vocabulary Σ, a vocabulary $\sigma \subseteq \Sigma$ with the same set of sorts, and a finite σ-structure I_σ, the *model expansion search problem* for input $\langle \Sigma, T, \sigma, I_\sigma \rangle$ is the problem of finding models M of T that expand I_σ, i.e., $M|_\sigma = I_\sigma$. The *MX($\mathcal{L}$) decision problem* is the problem of deciding whether such a model exists.

The vocabulary σ is called the *instance vocabulary* of the problem, the vocabulary $\Sigma \setminus \sigma$ the *expansion vocabulary*. I_σ is called the *instance structure*.

Observe that if T is a theory over a vocabulary Σ containing no function symbols, Herbrand model generation for T can be simulated by MX. Indeed, let σ be the set of constants in Σ, the domain of I_σ the Herbrand universe and $c^{I_\sigma} = c$ for every constant $c \in \sigma$.

On the other hand, when $\sigma = \Sigma$ solving the MX decision problem boils down to model checking.

The following are two examples of MX(FO(ID)) representations of well-known computational problems.

Example 1 (Graph Colouring). The instance vocabulary consists of two sorts, Vtx and $Colour$, representing respectively the vertices of the given graph and the colours. It also contains a predicate symbol $Edge$ with sort (Vtx, Vtx), representing the edges of the given graph. The expansion vocabulary consists of a single function symbol $Colouring$ of sort $(Vtx, Colour)$, representing the solution. The only sentence in the theory is $\forall v_1, v_2 \ (Edge(v_1, v_2) \supset Colouring(v_1) \neq Colouring(v_2))$.

For the instance structure I_σ given by $Vtx^{I_\sigma} = \{a; b; c\}$, $Colour^{I_\sigma} = \{blue; red\}$ and $Edge^{I_\sigma} = \{a, b; b, c\}$, a sample solution to the MX search problem is the structure M, expanding I_σ with $Colouring^M(a) = blue$, $Colouring^M(b) = red$ and $Colouring^M(c) = blue$.

Example 2 (Hamiltonian Path). The instance vocabulary contains a sort Vtx, a predicate symbol $Edge$ of sort (Vtx, Vtx) and a constant $Start$ of sort Vtx, which represents the first vertex in the path. The expansion vocabulary contains a predicate In and $Reached$, where $\mathfrak{s}(In) = (Vtx, Vtx)$ and $\mathfrak{s}(Reached) = Vtx$. In represents the edges that are in the path. The theory is given by

$$\forall v_1, v_2 \ (In(v_1, v_2) \supset Edge(v_1, v_2)).$$
$$\forall v_1, v_2, v_3 \ (In(v_1, v_2) \wedge In(v_1, v_3) \supset v_2 = v_3).$$
$$\forall v_1, v_2, v_3 \ (In(v_1, v_3) \wedge In(v_2, v_3) \supset v_1 = v_2).$$
$$\forall v \ \neg In(v, Start).$$
$$\forall v \ Reached(v).$$
$$\left\{ \begin{array}{l} \forall v \ Reached(v) \leftarrow v = Start. \\ \forall v \ Reached(v) \leftarrow Reached(w) \wedge In(w, v). \end{array} \right\}$$

Grounding

Solving the MX(FO(ID)) search or decision problem for input $\langle \Sigma, T, \sigma, I_\sigma \rangle$ can be done by creating an "equivalent" propositional theory T_g using T and I_σ and subsequently calling a model generator (in case of the search problem) or satisfiability checker (in case of the decision problem) for the propositional fragment of FO(ID). For solving the MX(FO(ID)) decision problem, it suffices that T_g is satisfiable iff T has a model expanding I_σ. For solving the search problem, a one-to-one correspondence between the models of T_g and the models of T expanding I_σ is required. Because GIDL is meant to be a grounder for the search problem, we consider the latter, stronger type of equivalence in this paper.

We now define grounding formally. Let σ be a subvocabulary of Σ with the same set of sorts and let I_σ be a σ-structure. Denote by Σ^{I_σ} the vocabulary Σ, extended with a new constant symbol $\mathbf{d}$ for every $d \in s^{I_\sigma}$, $s \in \Sigma_S$. We call these new constants *domain constants* and denote the set of all domain constants by $D(I_\sigma)$. For a Σ-structure M expanding I_σ, denote by $M^{D(I_\sigma)}$ the structure expanding M to Σ^{I_σ} by interpreting every $\mathbf{d} \in D(I_\sigma)$ by the corresponding domain element d. A formula is in *ground normal form* (GNF) if it contains no quantifiers and all its atomic subformulas are of the form $P(\mathbf{d}_1, \ldots, \mathbf{d}_n)$, $F(\mathbf{d}_1, \ldots, \mathbf{d}_n) = \mathbf{d}$, $c = \mathbf{d}$ or $\mathbf{d}_1 = \mathbf{d}_2$, where P, F and c are respectively a predicate, function and constant symbol of Σ, and $\mathbf{d}_1, \ldots, \mathbf{d}_n, \mathbf{d}$ are domain constants of the appropriate sorts. Observe that a GNF formula is essentially propositional.

A rule is in GNF if its body is in GNF and its head is of the form $P(\mathbf{d}_1, \ldots, \mathbf{d}_n)$, where $\mathbf{d}_1, \ldots, \mathbf{d}_n$ are domain constants.

Definition 2 (Grounding). Let T be a theory over Σ, $\sigma \subseteq \Sigma$ and I_σ a σ-structure. A *grounding for T with respect to I_σ* is a theory T_g over Σ^{I_σ} such that all sentences and rules occurring in T_g are in GNF and for every Σ-structure M expanding I_σ, $M \models T$ iff $M^{D(I_\sigma)} \models T_g$. T_g is called *reduced* if it contains no symbols of σ.

Input and Output Language

In this section, the input and output language of GIDL are described. The input language is called FO$^+$ and is an extension of FO(ID) with partial functions, subsorts, arithmetic and aggregates. The concrete syntax accepted by the system is basically an ASCII version of the input language as described below and can be found in the user manual of the system (Wittocx & Mariën 2008). The manual also describes the output syntax.

Basic Input

The input for GIDL reflects the input of an MX search problem. I.e., it consists of a declaration of an instance vocabulary σ, a sorted expansion vocabulary $\Sigma \setminus \sigma$, a theory T over Σ and a finite σ-structure. These four parts are separated by different headers and can be placed in different files if necessary. GIDL supports full FO(ID), i.e., T can contain arbitrary definitions, the same predicate can be defined in multiple definitions, terms can be nested arbitrarily deep, etc.

The variables occurring in T do not have to be declared. Their associated sort can be specified at the moment they are used in T. Moreover, GIDL contains a sort inference mechanism that derives the sort of a variable automatically if there is one and only one possibility for its sort such that a well-sorted formula is obtained[1].

The declaration of the expansion vocabulary can be split in a set of auxiliary symbols and a set of symbols whose interpretation is relevant to the solution of the problem. This information is passed to the propositional solver, such that it can report to the user only the interpretation of the latter symbols in the models it finds.

FO$^+$

We now describe the extensions of FO(ID) included in FO$^+$.

Partial Functions In standard FO and FO(ID), all functions are total. Besides total functions, one can also declare and use *partial* functions in GIDL. When declaring a partial function, it is possible to specify a domain where it is total.

In general, arbitrary use of partial function symbols creates an ambiguity problem. E.g., consider the formula $P(F(\bar{t}))$, where F is a partial function symbol. This formula can be interpreted in two different ways, as illustrated by the following non-ambiguous rewritings of it:

$$\exists y\ (F(\bar{t}) = y \wedge P(y)) \tag{1}$$

$$\forall y\ (F(\bar{t}) = y \supset P(y)) \tag{2}$$

Here, the atoms $F(\bar{t}) = y$ should be interpreted as $G_F(\bar{t}, y)$, where G_F denotes the graph of F. When F is total, both rewritings are equivalent, but this is not the case when F is partial. Indeed, for an interpretation I such that $\bar{t}^I$ is not in the domain of F^I, $I \not\models$ (1), but $I \models$ (2).

A simple solution to this ambiguity problem is to impose the syntax restriction that a partial function symbol F can only occur in atoms of the form $F(t_1, \ldots, t_n) = t_{n+1}$, where $t_1, \ldots, t_{n+1}$ are terms containing no partial function symbols. For such formulas, there is no ambiguity problem. GIDL does not impose this syntax restriction. Instead, it interprets positive occurrences of atoms $P(F(\bar{t}))$ by (2) and negative occurrences by (1). In other words, it assumes the interpretation where the truth of the sentences in T is maximized, while the truth of the rule bodies is minimized. In case this does not reflect the intended interpretation, a user has to write the sentences and definitions of T in a non-ambiguous form.

Partial functions can be declared by the user but are required also for a logically correct treatment of functions declared over subsorts and of partial arithmetic functions such as $\div$ and mod.

Subsorts In the vocabulary declaration part of an input for GIDL, one can specify that a sort s_1 is a direct subsort of at most one other sort s_2. In that case, the domain $s_1^{I_\sigma}$ of s_1 in the instance structure I_σ has to be a subset of $s_2^{I_\sigma}$. The corresponding hierarchy of sorts must be a collection of trees. The root of a tree in the hierarchy is called a *base sort*. By $\mathrm{base}(s)$, we denote the root of the tree where s occurs, i.e., the base sort above s.

In a context where subsorts are used, a formula is well sorted if the following hold:

- for each term $F(t_1, \ldots, t_n)$ where $\mathfrak{s}(F) = (s_1, \ldots, s_{n+1})$, $\mathrm{base}(s_i) = \mathrm{base}(\mathfrak{s}(t_i))$ for $1 \leq i \leq n$;
- for each atom $P(t_1, \ldots, t_n)$ where $\mathfrak{s}(P) = (s_1, \ldots, s_n)$, $\mathrm{base}(s_i) = \mathrm{base}(\mathfrak{s}(t_i))$ for $1 \leq i \leq n$;
- for each atom $t_1 = t_n$, $\mathrm{base}(\mathfrak{s}(t_1)) = \mathrm{base}(\mathfrak{s}(t_2))$.

A rule with head $P(t_1, \ldots, t_n)$ and $\mathfrak{s}(P) = (s_1, \ldots, s_n)$ is well-sorted if its body is well-sorted and $\mathfrak{s}(t_i) = \mathfrak{s}(s_i)$ for $1 \leq i \leq n$.

A function with sort $(s_1, \ldots, s_{n+1})$ is treated as a partial function whenever one of the input sorts $s_1, \ldots, s_n$ is not a base sort. For an interpretation I and an atom $P(t_1, \ldots, t_n)$ with $\mathfrak{s}(P) = (s_1, \ldots, s_n)$, we define $I \not\models P(t_1, \ldots, t_n)$ if for at least one i, $t_i^I \notin s_i^I$. This fixes the semantics for inputs with subsort declarations.

Whenever a variable x occurs in two positions with a different sort, e.g in $P(x)$ and in $Q(x)$, where $\mathfrak{s}(P) \neq \mathfrak{s}(Q)$, GIDL does not automatically derive a sort for x, as this can lead to unexpected situations. Instead, the user is then forced to declare the sort of the variable.

Arithmetic Besides the vocabulary specified by the user, the instance vocabulary σ of a GIDL input implicitly contains a sort int and the arithmetic functions $+$, $-$, $\cdot$, $\div$, $abs(\cdot)$ and mod. In every instance structure over σ, int is interpreted by the integers $\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$, $+$ by addition on $\mathbb{Z}$, $-$ by subtraction, $\cdot$ by multiplication, $\div$ by integer division, abs by the absolute value and mod by the remainder. Note that $\div$ and mod are partial functions on $\mathbb{Z}$ with domain $\mathbb{Z} \setminus \{0\}$. Terms of the form $t_1 + t_2$, $t_1 \cdot t_2$, etc, are of sort int.

To ensure that the grounding produced by GIDL is finite, the use of int is restricted, both in the vocabulary declaration and the theory. In the input and expansion vocabulary declaration, a sort can be declared to be a subsort of int and a variable may have sort int. On the other hand, predicate or function declarations with sort $(\ldots, int, \ldots)$ are not allowed. If in the theory, a variable x of sort int is universally, respectively existentially quantified, it should occur as $\forall x\ (\varphi \supset \ldots)$, respectively $\exists x\ (\varphi \wedge \ldots)$ where φ is a formula for which there exists a finite interval such that $M[x/d] \not\models \varphi$ for any model M of the theory and d outside that interval.

[1] Some of the language extensions described below allow for situations where there is more than one possibility to obtain a well-sorted formula.

We call φ a *bound* for x. GIDL requires that the bounds have a very simple form. E.g., an atom $P(\ldots, x, \ldots)$ is a bound. A formula $t_1 \leq x \leq t_2$ is a bound if t_1, respectively t_2, is a term for which there exists an $n_1 \in \mathbb{Z}$, resp. $n_2 \in \mathbb{Z}$ such that for each model M of the theory $n_1 \leq t_1^M$, respectively $n_2 \geq t_2^n$. Etc. Besides occurrences of bounds φ as $\forall x\ (\varphi \supset \ldots)$ or $\exists x\ (\varphi \wedge \ldots)$, GIDL also accepts syntactically equivalent forms like $\forall x\ (\ldots \vee \neg\varphi \vee \ldots)$ or $\exists x\ (\ldots \wedge \varphi \wedge \ldots)$.

Aggregates Aggregates are functions that have a set as argument. GIDL supports three aggregates: cardinality, sum and product. Concretely, the following are terms with sort int in the input language of GIDL: $card\{\overline{y} \mid \varphi[\overline{y}, \overline{z}]\}$, $sum\{x, \overline{y} \mid \varphi[x, \overline{y}, \overline{z}]\}$ and $prod\{x, \overline{y} \mid \varphi[x, \overline{y}, \overline{z}]\}$. The variables $\overline{z}$ are free in the aggregate term, while x and $\overline{y}$ are local to the term. The sort of x must be a subsort of int. Given an interpretation I, these terms are interpreted by

- $(card\{\overline{y} \mid \varphi[\overline{y}, \overline{z}]\})^I$ is the number of $\overline{d}$ such that $I[\overline{y}/\overline{d}] \models \varphi$;
- $(sum\{x, \overline{y} \mid \varphi[x, \overline{y}, \overline{z}]\})^I = \sum_{I[x/d_x, \overline{y}/\overline{d}_y] \models \varphi} d_x$;
- $(prod\{x, \overline{y} \mid \varphi[x, \overline{y}, \overline{z}]\})^I = \prod_{I[x/d_x, \overline{y}/\overline{d}_y] \models \varphi} d_x$;

Aggregates can be used everywhere in sentences or rule bodies where a term with a subsort of int can occur. The semantics for definitions containing recursion involving aggregates is the one presented in (Pelov, Denecker, & Bruynooghe 2005)

Example 3. In a machine scheduling problem, the constraint that at each timepoint t, the sum of the capacities c of the machines m that are not in maintenance must exceed 100 can be expressed by the sentence $\forall t\ (sum\{c, m \mid Capacity(m) = c \wedge \neg Maintenance(m, t)\} \geq 100)$.

Output Language

The output language of GIDL is an extension with rules and aggregates of the CNF format for SAT solvers and is called *extended CNF* (ECNF). It is the input format for the propositional solvers MIDL (Mariën, Wittocx, & Denecker 2007) and MINISAT(ID) (Mariën *et al.* 2008). Details about the syntax and semantics of the ECNF format is available at `www.cs.kuleuven.be/~dtai/krr/software.html`.

Translation Information

In an ECNF file, each propositional atom has a number, but not a name. In order to construct human readable solutions, GIDL also passes a *translation table* to the propositional solver, defining a mapping from each number that occurs in its ECNF output to a name. To avoid an exhaustive table mapping each number to its corresponding name, first all sort names and their domain elements are listed. Then all predicates with their corresponding sorts are listed, and are assigned a number. An atom $P(d_1, d_2)$ then corresponds to the number $n_P + (i_1 - 1) \cdot |s_2| + (i_2 - 1)$, where $\mathfrak{s}(P) = (s_1, s_2)$, n_P is the number assigned to P, d_1 the i_1th domain element of sort s_1, d_2 the i_2th domain element in s_2 and $|s_2|$ the size of the domain of s_2. The offsets n_P are chosen such that the numbers associated to atoms of different predicates do not overlap.

True and Arbitrary Atoms

Atoms that do not occur in an ECNF file are standard considered to be false by solvers. However, it is often desirable to also leave out the atoms that are discovered to be true in every model and the ones whose truth value can be arbitrarily chosen. GIDL passes a list of left out true and arbitrary atoms to the solver.

System Architecture

Given an input $\langle \Sigma, T, \sigma, I_\sigma \rangle$, GIDL constructs a grounding for T with respect to I_σ in six phases. In this section, a short description of each of the phases is given. The actual grounding algorithm (phase 5) is described in more detail in the next section.

Parser In the first phase, the input $\langle \Sigma, T, \sigma, I_\sigma \rangle$ is parsed. The parser of GIDL is implemented using *flex* and *bison*, which makes it easy to include future extensions of the input language.

Rewrite and Analyze In this phase, T is transformed into an internal normal form: negations are pushed inside until they are directly in front of atoms, $\supset$ is translated in terms of $\neg$ and $\vee$, functions are brought in the form $F(\overline{x}) = y$ and then, these atoms are replaced by $G_F(\overline{x}, y)$, where G_F is a new predicate representing the graph of F. Constraints are added to ensure that each G_F is a graph of a function. Also, all definitions are merged into a single definition Δ.

The dependency graph of Δ is constructed and analyzed to discover which defined predicates do not depend on open expansion predicates. The interpretation of these predicates is the same in every model of T expanding I_σ and can efficiently be computed. Also, a good grounding order for the rules of Δ is computed.

Pre-grounder The pre-grounder calculates the interpretation of the defined predicates that do not depend on open expansion predicates or on aggregates by evaluating their rules. The evaluation algorithm is a generalized version of the semi-naive technique (Ullman 1988) and can handle recursion over negation. The predicates whose interpretation is calculated are from then on considered to be part of the instance structure σ. I_σ is extended by assigning the computed relations to these predicates.

Approximation In this phase, an *approximation* for each subformula in T is computed, using the anytime algorithm described in (Wittocx, Mariën, & Denecker 2008). The computed approximations are used to both reduce grounding size and time.

Formally, an approximation for a formula $\varphi[\overline{x}]$ is a pair of formulas $(\varphi_{ct}[\overline{y}], \varphi_{cf}[\overline{z}])$ over σ such that $\overline{y} \subseteq \overline{x}$, $\overline{z} \subseteq \overline{x}$, $T \models \forall \overline{x}\ (\varphi_{ct} \supset \varphi)$ and $T \models \forall \overline{x}\ (\varphi_{cf} \supset \neg\varphi)$. Intuitively, the formula φ_{ct} provides a lower bound on the set of instances $\varphi[\overline{x}/\overline{\mathbf{d}}]$ of φ that are true in every model of T. The grounding algorithm can then safely replace instances $\varphi[\overline{x}/\overline{\mathbf{d}}]$ in this lower bound by $\top$, leading to a smaller

grounding. Vice versa, φ_{cf} provides a lower bound on the set of $\varphi[\overline{x}/\overline{\mathbf{d}}]$ that are false in every model of T. Instances $\varphi[\overline{x}/\overline{\mathbf{d}}]$ in this lower bound can be replaced by $\bot$. Observe that $(\bot, \bot)$ is an approximation for every formula, called the *trivial approximation*.

Example 2 (Continued). In the Hamiltonian path example, $(\bot, v_2 = Start \vee \neg Edge(v_1, v_2))$ is an approximation for the subformula $In(v_1, v_2)$ and $(Edge(v_1, v_2), \neg Edge(v_1, v_2))$ is an approximation for $Edge(v_1, v_2)$.

The maximal running time of the approximation algorithm, as well as the maximal size of the derived bounds can be specified by the user. Experiments in (Wittocx, Mariën, & Denecker 2008) showed that the default settings work well in most cases.

In the implementation, the approximations are represented and simplified using binary decision diagrams for FO as defined in (Goubault 1995). We extended the simplification algorithm of that paper with rules to cope with arithmetic. Also, parts of approximations that contain no free variables are evaluated out using the instance structure I_σ. This evaluation is the only part of the approximation algorithm that depends on I_σ.

Grounder Using the computed approximations for each subformula, an ECNF theory, equivalent to the input T and I_σ is constructed.

Translate Finally, the translation information and the list of true and arbitrary atoms is written to the output.

Grounding

The actual grounding component in GIDL accomplishes two tasks. It instantiates variables by domain elements and at the same time transforms complex formulas and rules into the ECNF format by applying the Tseitin transformation (Tseitin 1968). In this section, we present the grounding algorithm for the FO part of the input. The algorithm for grounding the rules is similar.

Procedure `Ground` gets as input a formula $\varphi[\overline{x}]$ and outputs a GNF theory, equivalent to the theory containing the single sentence $\bigwedge_{\overline{\mathbf{d}}} \varphi[\overline{x}/\overline{\mathbf{d}}]$. I.e., it outputs a grounding for the sentence $\forall \overline{x}\ \varphi[\overline{x}]$. Here, φ is assumed to be in the internal normal form of GIDL, i.e., the negations are in front of the atoms and function symbols F only occur in atoms of the form $F(\overline{y}) = z$. $(\varphi_{ct}, \varphi_{cf})$ denotes the approximation of φ.

The procedure `output` writes a single ground formula or rule to the output.

The function `getLit` implements the Tseitin transformation. It gets as input a formula $\varphi[\overline{x}]$, outputs a definition Δ_φ in GNF and returns a literal $P_\varphi \in \mathrm{Def}(\Delta_\varphi)$ such that in every model of Δ_φ, the truth value of P_φ equals the truth value of $\bigvee_{\overline{\mathbf{d}}} \varphi[\overline{x}/\overline{\mathbf{d}}]$. Our actual implementation of `getLit` involves some bookkeeping to make sure Δ_φ is written only once, even if `getLit(`φ`)` is called multiple times.

The purpose of line 15 of procedure `Ground`, is to compute all values $\overline{\mathbf{d}}$ such that $\bigvee_i \psi_i[\overline{x}/\overline{\mathbf{d}}]$ is not certainly true, i.e., to compute the answers of the conjunctive formula $\bigwedge_i \neg(\psi_i)_{ct}[\overline{x}]$ in I_σ. GIDL uses the backjumping algorithm of (Leone, Perri, & Scarcello 2004). The original algorithm was designed for computing answers to conjunctions of *literals* but, since the answers of the formulas $\neg(\psi_i)_{ct}[\overline{x}]$ can be easily computed as a table, it is easy to extend the algorithm.

To obtain a grounding of the FO part of T, `Ground` is applied on all sentences of T.

Procedure `ground(`φ`)`

```
1  if φ_cf ≡ ⊤ then
2      output ⊥; return ;
3  if φ_ct ≡ ⊤ then
4      output ⊤; return ;
5  Let x̄ be the free variables of φ;
6  switch φ do
7      case φ is a literal
8          for all d̄ such that I_σ ⊭ φ_ct[x̄/d̄] do
9              if I_σ ⊨ φ_cf[x̄/d̄] then
10                 output ⊥; return ;
11             else output φ[x̄/d̄];
12     case φ ≡ ⋀_{1≤i≤n} ψ_i
13         for 1 ≤ i ≤ n do ground(ψ_i)
14     case φ ≡ ⋁_{1≤i≤n} ψ_i
15         for all d̄ such that I_σ ⊭ ⋁_{1≤i≤n} (ψ_i)_ct[x̄/d̄] do
16             V := ∅;
17             for 1 ≤ i ≤ n do
18                 if I_σ ⊭ (ψ_i)_cf[x̄/d̄] then
19                     add getLit(ψ_i[x̄/d̄]) to V;
20             output ⋁_{L∈V} L;
21     case φ ≡ ∀y ψ
22         ground(ψ);
23     case φ ≡ ∃y ψ
24         for all d̄ such that I_σ ⊭ φ_ct[x̄/d̄] do
25             if I_σ[x̄/d̄] ⊨ φ_cf then
26                 output ⊥; return ;
27             else
28                 V := ∅;
29                 for all d′ such that
                   I_σ ⊭ ψ_cf[x̄/d̄][y/d′] do
30                     if I_σ ⊭ ψ_ct[x̄/d̄][y/d′] then
31                         add getLit(ψ[x̄/d̄][y/d′]) to
                           V;
32                 output ⋁_{L∈V} L;
```

Complexity of `Ground`

When all subformulas of a formula φ are assigned the trivial approximation $(\bot, \bot)$, applying `Ground` to φ consists

```
Function getLit(φ)
1  Let x̄ be the free variables of φ;
2  switch φ do
3      case φ is a literal
4          V := ∅;
5          for all d̄ such that I_σ ⊭ φ_cf[x̄/d̄] do
6              if I_σ ⊭ φ_ct[x̄/d̄] then
7                  add φ[x̄/d̄] to V;
8          if V is a singleton {P} then return P;
9          else
10             Let P be a new propositional atom;
11             output P ← ⋁_{L∈V} L;
12             return P;
13     case φ ≡ ∃y ψ
14         return getLit(ψ);
15     ⋮ // Other cases
```

of simply substituting the variables of φ by all possible domain constants of the appropriate sorts. Hence in this case, computing `Ground`(φ) takes time $\mathcal{O}(\prod_{s\in\Sigma_S}|s|^{n_s})$, where n_s is the number of variables of sort s in φ and $|s|$ the size of the domain of s^{I_σ} of s.

In the case arbitrary approximations are assigned to the subformulas of φ, the result of `Ground`(φ) will become smaller. On the other hand, the worst-case time complexity of computing `Ground`(φ) is then $\mathcal{O}(\prod_{s\in\Sigma_S}|s|^{n_s+d_s})$, where d_s is the number of variables of sort s that occur non-free in an approximation of a subformula of φ. This shows that grounding in the presence of non-trivial approximations may increase the complexity. In practice however, the approximations computed by the algorithm of (Wittocx, Mariën, & Denecker 2008) almost never slow down grounding. Instead, experiments in that paper show that they often lead to a dramatic speed-up.

Example 2 (Continued). Let φ be the formula $\neg In(x, y)$. If the approximation for φ is $(\bot, \bot)$, then `Ground`(φ, φ) takes time $\mathcal{O}((Vtx^{I_\sigma})^2)$. If the approximation is $(y = Start \vee \neg Edge(x, y), \bot)$, it takes only time $\mathcal{O}(|Edge^{I_\sigma}|)$.

Related Work

MXidL

A non-native approach to grounding MX(FO(ID)) consists of applying the algorithm presented in (Mariën, Gilis, & Denecker 2004) to transform an MX(FO(ID)) input into an equivalent normal logic program under the well-founded semantics. Then, a (slightly adapted) grounder for Answer Set Programming can be used to ground the logic program. This is the approach taken by MXIDL, the first implemented MX(FO(ID)) grounder. MXIDL supports full many-sorted FO(ID) and arithmetic, but no aggregates, subsorts and partial functions. Experiments with MXIDL were reported on in (Mariën, Wittocx, & Denecker 2006).

MXG

The first native grounding algorithm for MX(FO) and MX(FO(ID)) was described in (Patterson *et al.* 2006; 2007) and works on a table-by-table basis. I.e., to construct a grounding of a sentence φ, it proceeds by taking joins, projections, complements,... of the tables in the instance structure, ending up with a full grounding of φ. The algorithm in GIDL on the other hand, proceeds on a tuple-by-tuple basis. For every variable, it tries all the (relevant) substitutions by domain constants, and it outputs part of the grounding of φ as soon as possible.

An implementation of the grounding algorithm of (Patterson *et al.* 2006) was reported on in (Mitchell *et al.* 2006) and is called MXG. The MXG system implements only part of FO(ID). It allows only for definitions that do not depend on open expansion predicates and that do not involve recursion over negation. It does not support functions, subsorts or arithmetic.

Psgrnd

PSGRND (East *et al.* 2006) is a grounder for the extended logic of propositional schemata (East & Truszczynski 2006a). This logic is a restricted fragment of function-free FO, extended with cardinality aggregates. Also, it has restricted support for inductive definitions: each theory may contain one definition, and all rule bodies must be conjunctions of atoms.

PSGRND keeps the grounding in memory and performs unit propagation each time a clause is added to the grounding. As a post-processing step, it does a limited amount of forward checking on the grounding.

Answer Set Programming

Answer Set Programming (ASP) is a framework for declarative problem solving that is closely related to MX(FO$^+$). Answer set programs can be transformed into FO$^+$ theories in a modular way (East & Truszczyński 2006b). Moreover, the *structure* of ASP theories is the same as that of FO$^+$ theories and there are a lot of similarities between the methodology of modelling in ASP and in MX(FO$^+$) (Mariën, Gilis, & Denecker 2004; Mariën, Wittocx, & Denecker 2006).

On the other hand, there are several differences in the input languages for GIDL and ASP systems. E.g., GIDL allows for arbitrary FO$^+$ sentences and definitions, while an answer set program is basically one big definition, in which rule bodies are restricted to conjunctions of literals. An FO sentence φ is modelled in ASP by a rule with an empty head and body $\neg\varphi$, an open predicate can be modelled by defining it with a *choice rule*. Finally, the instance structure of an MX(FO$^+$) problem corresponds to a series of facts in an answer set program.

ASP systems work by grounding and propositional solving. Three ASP grounders are LPARSE (Syrjänen 1998), GrinGo (Gebser, Schaub, & Thiele 2007) and the grounding component of DLV (Dell'Armi *et al.* 2004). The algorithm in LPARSE works table-by-table, the algorithms in the other two grounders tuple-by-tuple.

Table 1: Impact of approximation (time)

	no approx.	4/4	5/8	6/64
15puzzle	4.89	3.70	3.63	9.00
bounded spanningtree	256.93	10.88	8.41	21.21
clique	1.33	2.36	2.45	###
blocked n-queens	22.44	3.50	3.51	3.51
algebraic groups	7.20	7.38	7.86	###
hamiltonian path	21.37	0.04	0.03	0.19
sokoban	0.49	0.24	0.26	0.31
schur numbers	12.49	0.56	1.32	2.52
sudoku	1.00	0.70	1.08	###

Table 2: Impact of approximation (size)

	no approx.	4/4	5/8	6/64
15puzzle	1461007	1219751	1219751	1219375
bounded spanningtree	8857075	2255522	2255522	2255522
clique	353800	353800	353800	###
blocked n-queens	923822	15822	15822	15822
alg. groups	4001420	3931659	3870081	###
ham. path	8404074	5701	5701	5701
sokoban	95279	74878	74878	74878
schur numbers	64300	62369	51454	47733
sudoku	319795	178828	109267	###

Table 3: Impact of domain atoms

	GIDL	PSGRND	GrinGo	LPARSE	DLV
Ham. circuit (time)	0.54	0.52	17.41	5.25	21.27
Ham. circuit (size)	1.00	1.00	30.80	26.00	24.76

Table 4: MX(FO^+) problems (time)

	GIDL	MXG	MXIDL
25-queens	0.16	0.76	0.93
50-queens	1.72	7.90	22.42
75-queens	8.03	33.39	165.71
algebraic groups (size 8)	0.86	3.11	3.27
algebraic groups (size 10)	3.40	11.65	12.12
algebraic groups (size 12)	10.94	34.96	37.48
graph colouring (64980 nodes, 4 colours)	8.82	11.84	###
graph colouring (64980 nodes, 6 colours)	13.32	18.94	###
tower of hanoi (8 discs)	0.87	2.26	145.37
latin square (dim 30)	3.40	9.65	8.28
social golfer (24 players, 6 groups, 8 weeks)	0.47	1.68	1.88

Experiments

In this section, we evaluate the impact of the approximation phase and compare GIDL's performance to other grounders.

All experiments in this section were run on a C2D 3GHz machine with 2GB RAM. All times are in seconds and are averaged over five runs. There was a time-out (###) of 600 seconds for each run. To measure the size of a ground theory, we counted the number of propositional atoms in it.

When comparing to other grounders, we used the standard parameters for the approximation phase of GIDL (see below). The times for GIDL include the time needed for the pre-grounding and the approximation phase. More detailed information, including the used problem encodings, is available at `www.cs.kuleuven.be/~dtai/krr/software/gidl.html`.

Impact of Approximation

Impact of different settings To evaluate the impact of the approximation algorithm, we ran GIDL with different settings. The resulting grounding times and sizes are shown in Tables 1 and 2. In these tables, the first number of the setting is the number of times an approximating formula can be refined. The second number is a measure for the maximum size of the approximating formulas. Increasing these numbers makes the approximation process more expensive and the computed approximating formulas larger and potentially more precise. Due to the increased precision of these formulas, the subsequent grounding phase will produce smaller groundings. This phase may be faster or slower depending on whether the gain due to the smaller grounding dominates the cost of evaluating the larger approximating formulas. The default setting of GIDL is 4/4.

The tables show that the use of approximation yields (often drastically) better times and sizes, even with few refinements and small formula sizes. Only in two cases, grounding without approximation is slightly faster. As for the impact of the size of the parameters, we observe that the most precise setting (6/64) produces a substantially smaller grounding in only one case while it has two time-outs. These and other experiments showed that GIDL's default setting 4/4 provides a good trade-off.

Domain Atoms In general, encoding problems for ASP solvers involves carefully adding (semantically redundant) *domain atoms* to obtain fast grounding times and small grounding sizes. Due to the approximation algorithm, this is not needed when encoding problems for GIDL. Instead, adding redundant domain atoms to GIDL's input rather increases the running time. Because of its unit propagation, the same observation holds for PSGRND. This is illustrated by Table 3 which, in case of a Hamiltonian circuit problem, shows the ratios of the grounding time and size for an encoding without to that for an encoding with redundant domain atoms.

Comparison to MX(FO^+) grounders

In this section, we compare GIDL to the other existing grounders for fragments of MX(FO^+): MXG (version 0.16)[2] and MXIDL. The ASP grounder used as back-end for MXIDL in the experiments is an adaption of GrinGo (version 0.0.1). The encodings of the problems in the first category are exactly the same for each of the three systems. Most of them were taken from `www.cs.sfu.ca/research/groups/mxp/examples/index.html`. The grounding times are shown in Table 4. GIDL consistently outperforms the other MX(FO^+) grounders. Table 5 shows the number of literal instances in the resulting ground files. In general, GIDL produces the smallest groundings, MXIDL the largest ones.

[2]There exists a newer version of MXG, but it was not available at the time of the submission deadline.

Table 5: MX(FO$^+$) problems (size)

	GIDL	MXG	MXIDL
24-queens	50850	50850	52500
50-queens	411700	411700	418125
75-queens	1395050	1395050	1409375
algebraic groups (size 8)	783209	1048119	1054835
algebraic groups (size 10)	3171771	3998889	4014163
algebraic groups (size 12)	9852733	11941763	11971803
graph colouring (64980 nodes, 4 colours)	4141440	2590560	###
graph colouring (64980 nodes, 6 colours)	6991920	4665600	###
tower of hanoi (8 discs)	517570	610904	7152033
latin square (dim 30)	1545113	2430400	2514530
social golfer (24 players, 6 groups, 8 weeks)	366144	534144	510378

Comparison to PSGRND and ASP grounders

In this section, we compare GIDL to PSGRND (7 jul 2005[3]), LPARSE (1.0.17[4]), GrinGo (1.0.0) and DLV (11 oct 2007).

The grounding times of GIDL, PSGRND and the ASP grounders are shown in Table 6, the sizes in Table 7. The problems were chosen such that their encodings cover a wide range of different formulas and language constructs. Because the grounders take different input languages, it is not possible to compare their performance in an entirely objective manner. To nevertheless obtain an as fair as possible comparison, the encodings are similar for the different grounders (i.e., as far as possible, they are straightforward translations of each other) except that domain atoms are added to the ASP encodings where needed to avoid the excessively bad grounding times and sizes mentioned above. Only the encoding of the sokoban puzzle differs considerably among the grounders, because it involves complex statements with alternating quantifiers which are not directly expressible in ASP. For the n-queens instances, we tried the grounders on two different encodings. The first one contains an explicit definition of the concept of a diagonal on a chess board. Due to the use of arithmetic in the second encoding, it could not easily be translated to the input language for DLV.

For each of the problems, GIDL ranks first or second in grounding time. Only on the first version of n-queens and the smallest instance of the Hamiltonian circuit, one of the ASP grounders is (slightly) faster. PSGRND outperforms GIDL on 3 of the seven problems, being at most 7 times faster. It is also faster on the *even/odd* problem but cannot handle large instances of it. On the remaining 3 problems, GIDL outperforms PSGRND. It is at least 30 times faster on the first version of the n-queens problem and on the largest instance of the magic series.

The good results of GIDL and DLV on the first version of n-queens is due to their pre-grounding phase using the semi-naive evaluation technique. The other three grounders use a combination of grounding and unit propagation to compute the diagonals on the chess board. On the *even/odd* problems, the semi-naive evaluation seems less efficient.

[3]We switched the forward checking phase of PSGRND off, as this phase is not incorporated in the other grounders. In all problems selected here, grounding with lookahead leads to slower grounding times.

[4]The newest version of LPARSE (1.1.1) appears to be a lot slower than version 1.0.17 and exhibits the same segmentation faults on our experiments

Table 6: Comparison to PSGRND and ASP grounders (time)

	GIDL	PSGRND	GrinGo	LPARSE	DLV
100-queens-v1	8.83	###	16.67	15.09	9.31
125-queens-v1	19.51	###	35.92	35.58	18.78
150-queens-v1	38.24	###	74.61	71.96	34.37
250-queens-v2	1.38	0.21	1.46	###	-
500-queens-v2	5.66	0.82	6.93	###	-
750-queens-v2	12.78	2.03	19.78	###	-
graph col. (4 colours)	4.72	1.78	9.02	seg. fault	9.12
graph col. (6 colours)	6.14	2.50	12.28	seg. fault	15.05
graph col. (8 colours)	7.56	3.32	15.58	seg. fault	22.62
magic series (size 250)	0.62	7.02	###	###	###
magic series (size 500)	2.49	71.45	###	###	###
magic series (size 750)	5.73	###	###	###	###
Ham. circuit (500 nodes)	0.67	0.80	2.69	0.55	0.83
Ham. circuit (1000 nodes)	2.19	3.99	11.80	2.20	2.53
Ham. circuit (1500 nodes)	4.88	11.11	34.45	6.51	5.67
even/odd (0..10^6)	9.10	3.37	10.17	seg. fault	###
even/odd (0..$2 \cdot 10^6$)	18.96	error	20.34	seg. fault	###
even/odd (0..$3 \cdot 10^6$)	29.06	error	30.23	seg. fault	###
sokoban (20 steps)	3.28	2.44	11.66	8.08	9.39
sokoban (40 steps)	7.00	5.03	23.26	16.29	19.72
sokoban (60 steps)	10.76	7.64	36.04	24.54	30.31

Table 7: Comparison to PSGRND and ASP grounders (size)

	GIDL	PSGRND	GrinGo	LPARSE	DLV
100-queens-v1	1333400	###	1990200	2688100	2050100
125-queens-v1	2604250	###	3890875	5241375	3984500
150-queens-v1	4500100	###	6727800	9047150	6862650
250-queens-v2	252488	250496	250746	###	###
500-queens-v2	1004988	1000996	1001496	###	###
750-queens-v2	2257488	2251496	2252246	###	###
graph col. (4 col.)	1810800	1810800	2069644	###	3630240
graph col. (6 col.)	2716200	2716200	2975046	###	7004880
graph col. (8 col.)	3621600	3621600	3880448	###	11419200
magic series (250)	315005	15939253	###	###	###
magic series (500)	1255005	126253503	###	###	###
magic series (750)	2820005	###	###	###	###
Ham. circ. (500)	97036	128464	121073	124573	192572
Ham. circ. (1000)	242033	320967	302067	309067	481066
Ham. circ. (1500)	483059	641441	603119	613619	961618
even/odd (0..10^6)	0	0	2000003	###	###
even/odd (0..$2 \cdot 10^6$)	0	###	4000003	###	###
even/odd (0..$3 \cdot 10^6$)	0	###	6000003	###	###
sokoban (20 steps)	1940913	1153859	3155238	5679469	3085574
sokoban (40 steps)	4085433	2425099	6430318	11357089	6363194
sokoban (60 steps)	6229953	3696339	9705398	17034709	9640814

The good result of GIDL on the magic series problems stems from its output format, which allows to define a single ground set and use it in multiple aggregate expressions. The other grounders write out a set for each aggregate expression, yielding cubic grounding size instead of GIDL's quadratic size in case of the magic series problem.

For each of the problems, either PSGRND or GIDL has the smallest grounding size. This is due to their rich output languages, enabling compact representations of, e.g., aggregate expressions, and to, respectively, the unit propagation and approximation algorithm. The zero grounding size in the *even/odd* problems stems from the fact that both PSGRND and GIDL write true atoms in the translation information, and not in the actual grounding.

Conclusions

We presented a grounder for an extension of FO, in the context of model expansion. An important contribution of the system is that it supports a very rich input language, extending full FO with ordered sorts, inductive definitions, aggregates, arithmetic and partial functions. The input language

and core algorithm of the grounder were described. Despite its rich language, which makes GIDL the most complete MX grounder of the moment, our experiments show that GIDL is the fastest MX grounder for (extensions of) FO and is more robust and often faster compared to ASP grounders.

Acknowledgments

The input language syntax of GIDL was designed in collaboration with David Mitchell and Eugenia Ternovska. The adapted version of GrinGo was written by Sven Thiele.

References

Dell'Armi, T.; Faber, W.; Ielpa, G.; Leone, N.; Perri, S.; and Pfeifer, G. 2004. System description: Dlv with aggregates. In Lifschitz, V., and Niemelä, I., eds., *LPNMR*, volume 2923 of *Lecture Notes in Computer Science*, 326–330. Springer.

Denecker, M., and Ternovska, E. 2004. A logic of nonmonotone inductive definitions and its modularity properties. In Lifschitz, V., and Niemelä, I., eds., *Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'7)*.

Denecker, M. 2000. Extending classical logic with inductive definitions. In Lloyd et al., J., ed., *First International Conference on Computational Logic (CL'2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*, 703–717. Springer.

East, D., and Truszczynski, M. 2006a. Predicate-calculus-based logics for modeling and solving search problems. *ACM Trans. Comput. Log.* 7(1):38–83.

East, D., and Truszczyński, M. 2006b. Predicate-calculus based logics for modeling and solving search problems. *ACM Transactions on Computational Logic (TOCL)* 7(1):38 – 83.

East, D.; Iakhiaev, M.; Mikitiuk, A.; and Truszczynski, M. 2006. Tools for modeling and solving search problems. *AI Commun.* 19(4):301–312.

Enderton, H. B. 1972. *A Mathematical Introduction To Logic*. Academic Press.

Gebser, M.; Schaub, T.; and Thiele, S. 2007. Gringo : A new grounder for answer set programming. In Baral, C.; Brewka, G.; and Schlipf, J. S., eds., *LPNMR*, volume 4483 of *Lecture Notes in Computer Science*, 266–271. Springer.

Goubault, J. 1995. A bdd-based simplification and skolemization procedure. *Logic Journal of IGPL* 3(6):827–855.

Kautz, H. A., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic and stochastic search. In *AAAI/IAAI, Vol. 2*, 1194–1201.

Leone, N.; Perri, S.; and Scarcello, F. 2004. Backjumping techniques for rules instantiation in the DLV system. In *NMR*, 258–266.

Marek, V. W., and Truszczyński, M. 1998. Stable models and an alternative logic programming paradigm. *CoRR* cs.LO/9809032.

Mariën, M.; Wittocx, J.; Denecker, M.; and Maurice, B. 2008. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In *Proceedings of the 11th conference on Theory and Applications of Satisfiability Testing, SAT 2008*, volume 4996 of *Lecture Notes in Computer Science*, 211–224. Springer.

Mariën, M.; Gilis, D.; and Denecker, M. 2004. On the relation between ID-Logic and Answer Set Programming. In Alferes, J. J., and Leite, J. A., eds., *JELIA'04*, volume 3229 of *Lecture Notes in Computer Science*, 108–120. Springer.

Mariën, M.; Wittocx, J.; and Denecker, M. 2006. The IDP framework for declarative problem solving. In *Search and Logic: Answer Set Programming and SAT*, 19–34.

Mariën, M.; Wittocx, J.; and Denecker, M. 2007. MidL: A SAT(ID) solver. In *4th Workshop on Answer Set Programming: Advances in Theory and Implementation*, 303–308.

Mitchell, D., and Ternovska, E. 2005. A framework for representing and solving NP search problems. In *AAAI'05*, 430–435. AAAI Press/MIT Press.

Mitchell, D. G., and Ternovska, E. 2008. Expressive power and abstraction in ESSENCE. *Constraints* 13(3).

Mitchell, D.; Ternovska, E.; Hach, F.; and Mohebali, R. 2006. Model expansion as a framework for modelling and solving search problems. Technical Report TR2006-24, Simon Fraser University.

Patterson, M.; Liu, Y.; Ternvska, E.; and Gupta, A. 2006. Grounding for model expansion in k-guarded formulas. In *Proceedings of 21st IEEE Symposium on Logic in Computer Science (LICS06)*.

Patterson, M.; Liu, Y.; Ternovska, E.; and Gupta, A. 2007. Grounding for model expansion in k-guarded formulas with inductive definitions. In Veloso, M. M., ed., *IJCAI*, 161–166.

Pelov, N.; Denecker, M.; and Bruynooghe, M. 2005. Well-founded and stable semantics of logic programs with aggregates. *CoRR* abs/cs/0509024.

Syrjänen, T. 1998. Implementation of local grounding for logic programs with stable model semantics. Technical Report B18, Digital Systems Laboratory, Helsinki University of Technology.

Tseitin, G. S. 1968. On the complexity of derivation in propositional calculus. In Slisenko, A. O., ed., *Studies in Constructive Mathematics and Mathematical Logic II*, volume 8 of *Seminars in Mathematics: Steklov Mathem. Inst.* New York: Consultants Bureau. 115–125.

Ullman, J. D. 1988. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press.

Van Gelder, A.; Ross, K. A.; and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the ACM* 38(3):620–650.

Wittocx, J., and Mariën, M. 2008. The IDP system. Obtainable via `www.cs.kuleuven.be/~dtai/krr/software.html`.

Wittocx, J.; Mariën, M.; and Denecker, M. 2008. Grounding with bounds. In *AAAI'08*, 572–577. AAAI Press.