Detection and Exploitation of Functional Dependencies for Model Generation

BRÖES DE CAT AND MAURICE BRUYNOOGHE
Department of Computer Science, KU Leuven, Belgium
(e-mail: {broes.decat, maurice.bruynooghe}@cs.kuleuven.be)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract
Recent work in Answer Set Programming has integrated ideas from Constraint Programming. This has led to a new field called ASP Modulo CSP (CASP), in which the ASP language is enriched with constraint atoms representing constraint satisfaction problems. These constraints have a more compact grounding and are handled by a new generation of search algorithms. However, the burden is on the modeler to exploit these new constructs in his declarative problem specifications. Here, we explore how to remove this burden by automatically generating constraint atoms. We do so in the context of \textsc{FOL}(\cdot)\textsc{IDP}, a knowledge representation language that extends first-order logic with, among others, inductive definitions, arithmetic and aggregates. We uncover functional dependencies in declarative problem specifications with a theorem prover and exploit them with a transformation that introduces functions. Experimental evaluation shows that we obtain more compact groundings and better search performance.

1 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. Here, a partial input structure over a finite and known domain is extended into a complete structure that is a model of a given theory. These paradigms are studied and applied in the fields of Constraint Programming (CP) \cite{Apt2003}, Answer Set Programming \cite{Niemelä2006} and Knowledge Representation \cite{Baral2003}.

A state-of-the-art approach is to reduce the input theory, formulated in an expressive logic, in a model-equivalence preserving way to a theory in a fragment of the language supported by some search algorithm. Afterwards, this algorithm searches for models of the theory. For example, model generation/expansion for the language \textsc{FO}(\cdot) \cite{Denecker2008} is performed by reducing theories to the ground language \textsc{PC}(\cdot), for which efficient search algorithms are available. The term \textit{grounding} refers to both the reduction process and to its outcome; the 2-step approach is called \textit{ground-and-solve}.

A first generation of model expansion 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. To overcome this limitation, techniques from Constraint Programming have been incorporated, giving rise to the field of
Fig. 1. Workflow. In a first “offline” phase, the theory is used to detect functional dependencies and functions are introduced until no more can be found (or a time-out is reached). This is repeated in the “online” phase, now combined with the input structure. The transformed theory is then passed to the ground-and-solve algorithm.

ASP modulo CSP (CASP) (Ostrowski and Schaub 2012). Search algorithms have been developed that allow constraint atoms in the input. These atoms are ground instead of propositional, have (non-Herbrand) function terms as arguments, and stand for the constraints of a CSP problem (Lierler 2012; Gebser et al. 2009). This gives rise to more compact groundings that often also yield better propagation. Among those next generation systems are the solvers Clingcon (Ostrowski and Schaub 2012), Ezcsp (Balduccini 2011) and Constraint(ID) (De Cat et al. 2013).

As the original ASP language does not support (non-Herbrand) functions, ASP modulo CSP systems extend the language with special-purpose constraints with a more compact grounding (see (Cabalar 2013; Bartholomew and Lee 2012; Lifschitz 2012; Balduccini 2012; Cabalar 2011; Lin and Wang 2008) for approaches to enhance ASP with functions); to a certain degree, this is also the case for the language Zinc (Marriott et al. 2008). However, the user has to use these constructs in his declarative specifications in order to obtain the benefits of a more compact grounding and a better performance.

This paper explores to what extent this burden can be removed from the modeler. We uncover functional dependencies in declarative problem specifications with a theorem prover and exploit them with a transformation that introduces functions and, in the process, eliminates quantified variables. This results in a more compact grounding and more efficient search. We do this in the context of FO(·)_{IDP}, the language supported by the knowledge-base system IDP (Bogaerts et al. 2012), which extends FO with aggregate functions, inductive definitions, partial functions, types and arithmetic. The same ideas could be applied in the context of ASP languages. The analysis can be performed on theories both with and without input structure, giving rise to the workflow of Figure 1.

FO(·)_{IDP} supports both functions and predicates and users are free to use predicates when some of its arguments depend functionally on each other (out of preference, ignorance or because the theory is a translation of an ASP theory). Also, it may happen that a functional dependency only holds for a particular problem instance (e.g., a graph where each vertex has exactly one outgoing edge).

Example 1
Consider a scheduling application involving some events (events) to be planned, each exactly once, over a large period of time (time). A total order < on events is given. One
Detection and Exploitation of Functional Dependencies for Model Generation

possible constraint is that the planning of events has to follow their order. In FO(·), this can be represented as the theory consisting of the following sentences, with a grounding size of ||event||^2 × ||time||^2 (typically measured in number of ground atoms):

\[ \forall e : \exists t : \text{planned}(e, t), \]
\[ \forall e_1 e_2 t_1 t_2 : e_1 < e_2 \land \text{planned}(e_1, t_1) \land \text{planned}(e_2, t_2) \Rightarrow t_1 < t_2. \]

However, if we can prove that the second argument of planned depends functionally on the first, then planned can be replaced by a function symbol, say \( f_{\text{planned}} : \text{event} \mapsto \text{time} \). By equivalence preserving transformations, a theory with a grounding size of only ||event||^2 can then be obtained, namely \( \forall e_1 e_2 : e_1 < e_2 \Rightarrow f_{\text{planned}}(e_1) < f_{\text{planned}}(e_2) \).

The grounding contains constraint atoms \( f_{\text{planned}}(e_1) < f_{\text{planned}}(e_2) \). In CASP cling-con syntax, the constraint atom is written as \( f_{\text{planned}}(E1) < f_{\text{planned}}(E2) \).

The paper is organized as follows. In Section 2, FO(·) and necessary concepts are introduced. Next, we present the detection algorithm in Section 3, and the theory transformations in Section 4. In Section 5, experimental results are presented; we finish with related work and conclusions in Section 6.

2 Preliminaries

This paper makes use of FO(·)\textsuperscript{IDP}, a many-sorted logic that extends First-Order Logic (FO) with aggregate functions, arithmetic, inductive definitions and partial functions. An FO(·)\textsuperscript{IDP} vocabulary consists of types and typed predicate and function symbols. The signatures of \( n \)-ary predicates \( P \) and functions \( f \) are denoted, respectively, as \( P(T_1, \ldots, T_n) \) and \( f(T_1, \ldots, T_n) : T_{n+1} \). For each type \( T \), a predicate symbol \( T(T) \) exists, interpreted \textit{true} for all domain elements in type \( T \). An atom/term is badly-typed if at least one of its arguments is outside the interpretation of the declared type of that argument position. Badly-typed \textit{atoms} or \textit{terms} containing badly-typed terms are always interpreted \textit{false}\footnote{The negation of a badly-typed atom is true.}.

We assume familiarity with first-order logic. We often follow some conventions for symbols: we use \( a \) for an atom, \( t \) for a literal (an atom or its negation), \( x \) and \( y \) for variables, \( \pi \) for a tuple of variables, \( D \) for a set of domain elements, \( t \) for a term, \( \overline{t} \) for a tuple of terms, \( t::t \) for the concatenation of a tuple and a term, \( \varphi \) for a formula, \( c \) for a constant, \( f \) for a function and \( P \) and \( Q \) for predicates. With \( \overline{t} = ⟨t_1, \ldots, t_n⟩ \) a tuple and \( S = [s_1, \ldots, s_m] \) a subsequence of \( [1, n] \) (an index set), \( \overline{t}_S \) denotes the tuple \( ⟨t_{s_1}, \ldots, t_{s_m}⟩ \) while \( S^c \) denotes the complement of \( S \) with respect to \( [1, n] \), i.e., the elements of \( S \) are removed. Given two tuples \( \overline{t} \) and \( \overline{t}' \), both of length \( n \), \( \overline{t} = \overline{t}' \) denotes the conjunction \( t_1 = t'_1 \land \ldots \land t_n = t'_n \). With \( t[t' / t''] \) we denote a term \( t \) with occurrences of \( t' \) and with \( t[t'/t''] \) the term where these occurrences are replaced by \( t'' \).

A term is an FO term or an aggregate term. Aggregate terms consist of an aggregate function (cardinality, sum, product, minimum or maximum) and a set comprehension of the form \( \{ \pi : \varphi : t \} \), with \( \varphi \) a formula and \( t \) a term. It is interpreted as the aggregate function applied to the multiset of instances of \( t \), one for each distinct instance of \( \pi \) for which \( \varphi \) is true. The cardinality function, \( \#\{ \pi : \varphi : t \} \), counts the number of distinct instances of \( \pi \) that makes \( \varphi \) true and can be abbreviated as \( \#\pi \varphi \) as \( t \) is irrelevant.
The aggregate functions sum, product, min and max map the multiset of instances of $t$ to, respectively, the sum, product, minimum, and maximum of the multiset of terms. When the multiset is empty, the result is $0$, $1$, $+\infty$ and $-\infty$, respectively. For example, \(\text{sum}(\{x \ y : P(x, y) : f(x, y)\})\) is interpreted as the sum of the values of $f(x, y)$ for all instantiations of $x$ and $y$ for which $P(x, y)$ is true.

Functions are total unless declared as partial. An interpretation \(I\) is two-valued for a total (partial) function if, for every tuple of domain elements in its domain, it maps to exactly one (at most one) domain element in its codomain.\footnote{This deviates from the standard definition in order to take the type definition into account.} A formula \(\exists x : f(\overline{y}) = x\) is true iff $f$ has an image in its codomain (given \(I\)); it is abbreviated as $\text{HasImage}(f(\overline{y}))$.\footnote{A search algorithm with support for functions need not ground the formula for all values of $x$.} An atom $a$ containing a term $f(\overline{y})$ over a partial function symbol $f$ is interpreted as $\text{HasImage}(f(\overline{y})) \land a$; it is false when $f(\overline{y})$ has no image.

A function can be defined by a set of rules of the form $\forall \overline{x} : f(\overline{t}) = t' \leftarrow \varphi$, a predicate by rules $\forall \overline{x} : P(\overline{t}) \leftarrow \varphi$. These rule sets are called definitions and are interpreted according to the well-founded semantics (Van Gelder 1993).\footnote{An interpretation \(I\) satisfies the definition of an $n$-ary function $f$ if it satisfies its graph (i.e., the $n + 1$-ary relation corresponding to the function) and $f$ is functional in \(I\) (partial if $f$ is partial).} We assume rule sets for which the well-founded semantics are two-valued. The completion of the rules defining a symbol $s$ in definition $\Delta$ is denoted by $\text{comp}_\Delta(s)$.

A theory consists of sentences and definitions. The notation $\text{voc}(\mathcal{T})$ is used to denote the vocabulary of a theory $\mathcal{T}$. Besides a vocabulary and a theory, an FO\(\cdot\) specification also contains an input structure that interprets all types of the theory and (partially) interprets the functions and predicates. A typical computational task (model expansion) is to find an extension of the input structure that satisfies the sentences and is the two-valued well-founded model of the defined predicates and functions.

Given a vocabulary $\Sigma$, two theories $T$ and $T'$ are $\Sigma$-equivalent if each model of $T$ restricted to $\Sigma$ can be extended to a model of $T'$ and vice-versa. Two theories $T$ and $T'$ are strongly $\Sigma$-equivalent if the above extensions are also unique.

We assume theories are in flat negation normal form (FNNF): negations only occur directly in front of atoms, $\Rightarrow$ and $\Leftrightarrow$ are eliminated and a logical operator never occurs as a direct subformula of the same operator (e.g., $a_1 \lor a_2 \land a_3$ is in FNNF, but $a_1 \lor (a_2 \lor a_3)$ is not). This assumption is without loss of generality as any theory can be transformed into an equivalent FNNF theory in time polynomial in the size of the theory.

## 3 Detecting Functional dependencies

On the next page, the FO\(\cdot\) theory for the well-known packing-problem for squares is shown. It makes use of the predicates $\text{size}(\text{id}, \text{nb})$ and $\text{area}(\text{nb}, \text{nb})$ (interpreted in the input structure, together with the types $\text{id}$ and $\text{nb}$) and of the predicate $\text{pos}(\text{id}, \text{nb}, \text{nb})$ that specifies the $x$- and $y$-coordinate of the bottom-left corner of the square $\text{id}$. The sentences express respectively the constraints (as FO sentences) that (1) each square is placed at exactly one (the $\exists$ quantifier) position, (2) no squares overlap and (3) each square fits completely inside the specified area. The predicate $\text{noOverlap}/3$, used in (2), is defined between "[" and "]" and makes use of two auxiliary defined predicates.
The grounding of this theory can become very large. For example, rule (4) has a grounding size of $n^a b^2$, with $n$ the number of squares and $a$ and $b$ respectively the length and width of the area. However, $\text{pos}$, $\text{size}$ and $\text{area}$ in fact represent respectively 2, 1 and 2 functional relationships. So, for example, the body of (4) could be replaced by the formula $\text{pos}_x(id_1) + \text{size}(id_1) \leq \text{pos}_x(id_2)$ where $\text{pos}_x$ and $\text{size}$ are functions derived from the predicates $\text{pos}$ and $\text{size}$; this formula has a grounding size of only $n^2$.

A predicate has a functional dependency from a set of arguments $S$ (an index set) to an argument $j$ if a value for the arguments in $S$ uniquely determines the value of the argument $j$. For an $n-1$-ary function, the output can be considered as the $n^{th}$ argument; a function always has a dependency from the $n-1$ input arguments to the output argument, but also other dependencies can be present. A functional dependency can be formalised as a mapping from an index set to an argument position.

**Definition 1 (Functional dependency)**

Let $P(T_1, \ldots, T_n)/f(T_1, \ldots, T_{n-1}) : T_n$ be the signature of a predicate/function, $S$ an index set over $[1, n]$, $j$ an index in $S^c$, and $T$ a theory in which $P/f$ occurs. We have a **partial functional dependency** from $S$ to $j$ if, in each model $T$ of $\mathcal{T}$, it holds that for each tuple $\vec{d}_r \in (T_1 \times \ldots \times T_n)|_S$, all tuples $\vec{d}$ in $P^T/f^T$ for which $\vec{d}|_S = \vec{d}_r$ have the same value for $d'_j$ (uniqueness). We have a **total functional dependency** if, in addition, for each tuple $\vec{d}_r \in (T_1 \times \ldots \times T_n)|_S$, there is a tuple $\vec{d}$ in $P^T/f^T$ for which $\vec{d}|_S = \vec{d}_r$ (existence).

The uniqueness property expresses that a tuple in the index set $S$ maps to at most one value for the $j^{th}$ argument; the existence property that there is such a value for each well-typed tuple over $S$. Dependencies as above are denoted by $d(P(T_1, \ldots, T_n), S, j)/d(f(T_1, \ldots, T_{n-1}) : T_n, S, j)$, a subscript total (partial) is added to denote total (partial) dependencies whenever relevant. The index set $S$ and the argument $j$ are called, respectively, the domain and the codomain of the functional dependency. Note that in the case of functions, the codomain (index $n$) of the function can be part of the domain of a dependency. For example, a bijective function $f(T_1) : T_2$ has the dependencies $d(f(T_1) : T_2, \{1\}, 2) \text{ and } d(f(T_1) : T_2, \{2\}, 1)$.

**Proposition 1 (Function constraints)**

For an index set $S$ and index $j$, the existence property holds for a predicate $P(T)$ (a
function \( f(T) : T_n \) in a theory \( T \) iff \( T \) entails the existence constraint:

\[
\forall (x_i : i \in S) : \bigwedge_{i \in S} T_i(x_i) \rightarrow \exists (x_i : i \in S^c) : \bigwedge_{i \in S^c} T_i(x_i) \land P(T) \quad \text{(predicate)}.
\]

\[
\forall (x_i : i \in S) : \bigwedge_{i \in S} T_i(x_i) \rightarrow \exists (x_i : i \in S^c) : \bigwedge_{i \in S^c} T_i(x_i) \land f(T) = x_n \quad \text{(function)}.
\]

The uniqueness property holds iff \( T \) entails the uniqueness constraint:

\[
\forall x' : P(T) \land P(T') \land (x |_{S} = x' |_{S}) \Rightarrow x = x' \quad \text{(predicate)}.
\]

\[
\forall x' : f(T) = x_n \land f(T') = x'_n \land (x :: x_n) |_{S} = (x' :: x'_n) |_{S} \Rightarrow x = x' \quad \text{(function)}.
\]

In what follows, shorthands \( C_{\text{exists}}(P(T), S, j) \), respectively, \( C_{\text{exists}}(f(T) : T_n, S, j) \), \( C_{\text{unique}}(P(T), S, j) \) and \( C_{\text{unique}}(f(T) : T_n, S, j) \), are used for these constraints.

Proposition 2 ([Armstrong 1974])

For a theory \( T \), a total (partial) functional dependency of a position \( j \) on an index set \( S \) for a symbol in \( T \) implies a total (partial) functional dependency of \( j \) on all index sets of the given symbol that are supersets of \( S \) and do not contain \( j \).

This proposition offers two opportunities to prune the search. First, if one can prove

\[-C_{\text{unique}}(P(T), S, j) \text{ (or } -C_{\text{unique}}(f(T) : T_n, S, j))\text{, then there is no dependency of } S \text{ on } j \]

and one need not consider subsets of \( S \). Second, if a dependency from \( S \) to \( j \) is found, one need not consider supersets of \( S \). Our current implementation only exploits the latter and starts from the smallest candidate index sets (starting with \( \emptyset \), i.e., a constant argument). Each time one is found, the theory is rewritten (see next section) and detection continues on the new theory. However, that way, one likely never analyses the largest index sets (due to time-out), while dependencies involving them are quite frequent for predicates. So, for predicates, before exploring index sets from small to large, we first check for a dependency with an index set of size \( n - 1 \) and store it when found. Then we process index sets from small to large. If the algorithm aborts because of a time-out, the stored dependency, if present and not pruned, is used to rewrite the theory.

---

5 \( \{x_i : i \in S\} \) is a shorthand for \( (x_{s_1}, \ldots, x_{s_m}) \), with \( S = \{s_1, \ldots, s_m\} \).

6 Anytime algorithms can be halted at any point in time and return the best solution found till then.
As already mentioned, we use a theorem prover to check for the functional dependencies. Our current prototype uses SPASS ([Weidenbach et al. 2009]), a prover for theories in FO. Although the existence and uniqueness constraints are FO formulas, our theories however are in \( FO(\cdot) \), which includes definitions and aggregates. In some cases, definitions are equivalent to their completion (which are FO sentences); in other cases, the completion is a weaker theory. However, because a definition entails its completion and entailment is transitive, dependencies derived from the completion also hold for the original theory. Below, we describe two transformations from an \( FO(\cdot) \) theory to an FO theory. The first is equivalence preserving, the second produces a weaker theory. Note that standard FO is untyped and only supports total functions.

**Definition 2 (Strong FO(\( \cdot \))-to-FO transformation)**

The following rewrite rules are applied, in the given order:

1. For every \( n \)-ary predicate \( P(T_1, \ldots, T_n) \), type information is made explicit by adding the sentence \( \forall x : P(x) \Rightarrow T_1(x_1) \land \ldots \land T_n(x_n) \) (similarly for functions).
2. For every partial function \( f(T_1, \ldots, T_{n-1}) : T_n \) do:
   - (i) Introduce a new predicate symbol \( P_f(T_1, \ldots, T_n) \) and add the constraint \( C_{unique}(P_f(T_1, \ldots, T_n), [1, n-1], n) \) to \( T \).
   - (ii) Occurrences \( f(T) \) not at the top level of atoms \( f(T) = y \) with \( y \) a variable, are unnested by introducing a fresh variable.
   - (iii) Atoms of the form \( f(T) = y \), with \( y \) a variable, are replaced by \( P_f(T :: y) \).
3. Atoms \( \# \{ \pi : \varphi \} \geq n \), with \( n \) a natural number (cardinality aggregates with a known bound), are replaced by \( \exists \pi_1 \ldots \pi_n : (\land_{i,j\in[1,n]} i \neq j \pi_i \neq \pi_j) \land_{i\in[1,n]} \varphi[\pi_i/\pi] \). Other comparison operators are rewritten in a similar way.
4. Aggregates are transformed into FO as suggested in Section 5.4 of ([Pelov 2004]).
5. Definitions \( \Delta \) are rewritten using a voc(\( \Delta \))-equivalence preserving transformation based on ideas in ([Janhunen et al. 2009] Pelov and Trnovska 2005).
6. Formulas \( \forall x \in T : \varphi \) are replaced by \( \forall x : (\land_{i\in[1,n]} T_i(x_i)) \Rightarrow \varphi ; \) similarly for existential quantifications and set closures.

Details on cases 4 and 5 are in Appendix A. These two cases often result in large formulas (with lots of arithmetic) and hence can substantially increase the run-time of the theorem prover. Alternatively, a weak FO(\( \cdot \))-to-FO transformation can be applied. It is identical to the above one except that rule 3 is only applied for small \( n \) (e.g., \( n < 3 \)) and that rules 4 and 5 are replaced by the following rules.

**4w-a.** Unnest any aggregate term \( agg \) not in an atom \( agg = y \) with \( y \) a variable, by introduction of a fresh variable, confer rule 2.(ii).

**4w-b.** Replace any positive (negative) atom occurrence containing an aggregate term by true (false) if it is not in a definition; and by \( P(\pi) \) otherwise, with \( \pi \) the free variables of the atom and \( P \) a new predicate. For FNNF theories, this results in a weaker theory.

**5w.** Replace a definition by its completion.

**Proposition 3**

Let \( T_s \) be the strong and \( T_w \) be the weak FO(\( \cdot \))-to-FO transformation of \( T \). With the understanding that a partial function \( f_P \) in \( T \) corresponds to a predicate \( P_f \) in \( T_s \) and \( T_w \), it holds that: (i) a dependency is entailed by \( T \) if it is entailed by \( T_s \) and \( T_w \), and (ii) a dependency entailed by \( T_w \) or \( T_s \) on symbols in voc(\( T \)) is entailed by \( T \).
Proof sketch

(i) holds because rules 1, 3, 4, and 5 preserve strong equivalence. As for rule 2, a tuple \( \overline{a} \) is part of the interpretation of \( f \) in a model \( \mathcal{I} \) of \( \mathcal{T} \) iff it is part of the corresponding model of \( P_f \) in \( T_n \). (ii) holds because rule 4w-a preserves strong equivalence and rules 4w-b and 5w make sure that formulas are replaced by weaker formulas, hence models are preserved and only extra models can be created, so no new functional dependencies can be introduced by the transformation.

Example 2

Applying the transformation on the square-packing example replaces constraint (1) (in fact a cardinality constraint) by the sentences \( \forall \overline{x} p(id(id) \Rightarrow \exists x y : pos(id, x, y) \land \exists \overline{y} \overline{z} p(id, x, y_1) \land p(id, x, y_2) \Rightarrow x_1 = x_2 \land y_1 = y_2 \) (rule 3).

Rule 1 is applied to all predicates; e.g., for the predicate \( pos \), constraint (1) (in \( \mathcal{T} \)) is part of the interpretation of \( d \). Rule 2 is only applicable to \( \overline{y} \overline{z} p(id, x, y_1) \land p(id, x, y_2) \Rightarrow x_1 = x_2 \land y_1 = y_2 \) (rule 3).

\[ \forall \overline{x} (id(id) \Rightarrow \exists x y : pos(id, x, y) \land \exists \overline{y} \overline{z} p(id, x, y_1) \land p(id, x, y_2) \Rightarrow x_1 = x_2 \land y_1 = y_2 \] (rule 3).

4w-a and 5w make sure that formulas are replaced by weaker formulas, hence models are preserved and only extra models can be created, so no new functional dependencies can be introduced by the transformation.

The rewriting for a functional dependency \( d \left( \{ f(\overline{T}) \} : T_n, S, \overline{j} \right) \) starts with a pre-processing phase. Each rule of the form \( \forall \overline{x} : head[f(\overline{T})] \leftarrow \varphi \) is replaced by \( \forall \overline{x}, \overline{y} \in T_n : head[f(\overline{T})/x] \leftarrow y = f(\overline{T}) \land \varphi \). Also, if an aggregate term \( agg(\{ \overline{x} : \varphi \}) \) has an occurrence of \( f(\overline{T}) \) in \( t \) (or \( t = f(\overline{T}) \) then it is replaced by \( agg(\{ \overline{x}, y \in T_n : \varphi \land y = t : y \}) \).

Finally, if \( n \in S \), \( f \) is replaced by a predicate \( P_f \) as follows: \( a[f(\overline{T})] \), apart from

\[ \text{Definition 3 (dep-reduce)} \]

The rewriting for a functional dependency \( d \left( \{ f(\overline{T}) \} : T_n, S, \overline{j} \right) \) starts with a pre-processing phase. Each rule of the form \( \forall \overline{x} : head[f(\overline{T})] \leftarrow \varphi \) is replaced by \( \forall \overline{x}, \overline{y} \in T_n : head[f(\overline{T})/x] \leftarrow y = f(\overline{T}) \land \varphi \). Also, if an aggregate term \( agg(\{ \overline{x} : \varphi \}) \) has an occurrence of \( f(\overline{T}) \) in \( t \) (or \( t = f(\overline{T}) \) then it is replaced by \( agg(\{ \overline{x}, y \in T_n : \varphi \land y = t : y \}) \).

Redundant constraints can improve search performance.
Consider rule (4) of our packing problem: applying dep-reduce for the functional dependency $d_{\text{total}}(\text{pos}_y(id, \text{nb}), \{1\}, 2)$ introduces a function we rename as $\text{pos}_x(id) : \text{nb}$ and a relation $\text{pos}_x(id, \text{nb})$. For the new theory, another functional dependency can be proven, namely $d_{\text{total}}(\text{pos}_y(id, \text{nb}), \{1\}, 2)$. Again applying dep-reduce introduces $\text{pos}_y(id) : \text{nb}$.

After these two steps, rule (4) is rewritten into:

$$\{ \forall \text{id}_1 \text{id}_2 : \text{leftof}(\text{id}_1, \text{id}_2) \leftarrow \exists \text{x}_1 \text{y}_1 \text{x}_2 \text{y}_2 \text{s}_1 : \text{y}_1 = \text{pos}_y(\text{id}_1) \land \text{x}_1 = \text{pos}_x(\text{id}_1) \\ \land \text{size}(\text{id}_1, \text{s}_1) \land \text{y}_2 = \text{pos}_y(\text{id}_2) \land \text{x}_2 = \text{pos}_x(\text{id}_2) \land \text{x}_1 + \text{s}_1 \leq \text{x}_2 \}$$
While we have now replaced symbols by symbols of lower arity, we have not eliminated any variables. However, postprocessing can do so. For example the body of the rule of the above example can be simplified into \( \exists s_1 : \text{size}(id_1,s_1) \land \text{pos}_x(id_1) + s_1 \leq \text{pos}_x(id_2) \). The \( \text{FO}()^{\text{dep}} \) grounder, which aims at grounding human-written theories, does a poor job on such formulas. We preprocess the theory by the following set of equivalence preserving rewrite rules (applied on formulas in FNNF).

- The atom \( x = f(t) (\neg(x = f(t))) \) is a conjunct (disjunct) of a conjunct (disjunction) \( \varphi \). Replace \( x \) by \( f(t) \) in the other conjuncts (disjuncts) of \( \varphi \).
- A formula \( \text{agg}((\varphi, y : y = f(t) \land \varphi : y)) \). Replace it by \( \text{agg}((\varphi : \varphi : f(t))) \).
- A rule \( \forall a : a \leftarrow f(t) \). Replace \( a \) in \( f(t) \).
- A formula \( \exists x : \varphi \) and the only occurrence of \( x \) is in a conjunct \( x = f(t) \) of \( \varphi \). If \( f \) is total, remove the conjunct; otherwise replace it by \( \text{HasImage}(f(t)) \).
- A formula \( \forall x : \varphi \) and the only occurrence of \( x \) is in a disjunct \( \neg(x = f(t)) \) of \( \varphi \). If \( f \) is total, remove the disjunct; otherwise replace it by \( \neg\text{HasImage}(f(t)) \).
- A rule \( \forall x : a \leftarrow f(t) = x \land \varphi \) and the only occurrence of \( x \) is in \( f(t) = x \). If \( f \) is total, remove the conjunct; otherwise replace it by \( \text{HasImage}(f(t)) \).
- A formula \( \forall x : \varphi \) or \( \exists x : \varphi \) such that \( x \) does not occur in \( \varphi \). Replace it by \( \varphi \).

**Example 4**

Consider the problem of scheduling courses at a university. A naive modeler might use a symbol \( \text{planned}/5 \) to associate a session with a student group, a classroom, a time slot and a teacher all at once. The restriction that a teacher cannot teach multiple sessions at the same time might then be expressed by \( \forall \text{sid} \text{ sg c ts te : planned}(\text{sid}, \text{ sg}, c, \text{ ts}, \text{ te}) \Rightarrow \neg \exists \text{sid}_2 \text{ sg}_2 c_2 : \text{sid}_2 \neq \text{sid} \land \text{planned}(\text{sid}_2, \text{ sg}_2, c_2, \text{ ts}, \text{ te}) \), which has an impractical grounding size in the order of \( ||\text{sessions}||^2 \times ||\text{groups}||^2 \times ||\text{rooms}||^2 \times ||\text{slots}|| \times ||\text{teachers}|| \) atoms.

As all those relations are functional, function detection and rewriting will split \( \text{planned} \) in four function symbols and produce the sentence \( \forall \text{sid} \text{ sid}_2 : \text{sid} \neq \text{sid}_2 \land \text{teacher} \equiv \text{planned}(\text{sid}_2, \text{ sid}, \text{ teacher} \equiv \neg(\text{timeslot} = \text{timeslot}(\text{sid}_2)) \). This is the theory an experienced modeler would construct, but generated from the specification of an inexperienced user.

So far, we have only handled dependencies for non-defined symbols. When the symbol is defined, the rewriting \( \text{dep-reduce} \) is first applied to all atoms except the heads of rules. Afterwards, we have to replace those by rules for the new symbols introduced by \( \text{dep-reduce} \). This is achieved by the following definition which distinguishes several cases.

**Definition 5 (Define new symbols)**

- A dependency \( d(\langle f(T) : T_n, S, j \rangle) \) with \( j \neq n \) and \( n \notin S \). Each rule \( f(\pi) = x_n \leftarrow \varphi \) is replaced by \( f_d(\pi|_g) = x_j \leftarrow \varphi \) and \( f_r(\pi|_{(j)c}) = x_n \leftarrow \varphi \).
- A dependency \( d(\langle P(T) : T_n, S, j \rangle) \) with \( j = n \). Each rule \( f(\pi) = x_n \leftarrow \varphi \) is replaced by \( f_d(\pi|_g) = x_n \leftarrow \varphi \) and \( f_r(\pi|_{(j)c}) = x_j \leftarrow \varphi \).

\(^8\) In the rewriting, only atoms \( x = f(t) \) are used where the type of \( x \) is the output type of \( f \). If the type of \( x \) is empty, the atom is false.

\(^9\) \( \forall x : \neg(x = f(t)) \) is equivalent with \( \exists x : x = f(t) \) which is equivalent with \( \neg\text{HasImage}(f(t)) \).
Example 5
Consider a definition of nodes reachable from a given start node \( s \) and the minimal cost for reaching them, given a graph \( e(node, node) \) and a cost function \( c(node, node) : weight \):

\[
\begin{align*}
\forall x \ y \ c_o & : r(x, c_o + c(y, x)) \\
& \quad \land \forall y' \ c'_o : \neg(y = y') \land r(y', c'_o) \land e(y', x) \Rightarrow c(y, x) + c_o \leq c(y', x) + c'_o
\end{align*}
\]

The theory entails that the cost is a partial function on nodes, so can be rewritten as

\[
\begin{align*}
\forall x \ y : f_r(x) = 0 & \quad \leftarrow \forall x \ y : f_r(x) = f_r(y) + c(y, x) \\
& \quad \land \forall y' : \neg(y = y') \land e(y', x) \Rightarrow c(y, x) + f_r(y) \leq c(y', x) + f_r(y')
\end{align*}
\]

The current solver has no support for defined function symbols; however further transformations make exploitation possible. See Appendix C for details.

Integration within model expansion. When our theory rewriting is done as part of a model generation inference task, we cannot just replace a symbol with a set of symbols of lower arity. Indeed, the input structure uses the original alphabet and also the model should be presented to the user in the original alphabet. The rules given in Definition 4 provide the link between both alphabets. They are used during the grounding phase to translate the given input structure into corresponding input in the new alphabet. They are not needed during the solving (hence need not be grounded), but are used again to translate the model expressed in the new alphabet back to the original alphabet, which can also be done efficiently (without grounding) by bottom-up evaluation of the definitions.

5 Experiments

Our implementation uses the IDP system \cite{Bogaerts2012}, a knowledge-base system supporting state-of-the-art model expansion, as can be observed from previous ASP competitions \cite{Denecker2009} \cite{Calimeri2012}. As back-end, IDP uses the search algorithm \textsc{Constraint(ID)} \cite{DeCat2013}, which integrates \textsc{SAT} with, among others, unfounded set detection, aggregates and finite-domain constraints. To detect dependencies, we used the award-winning prover \textsc{SPASS} \cite{Weidenbach2009}. As benchmarks, we used faithful translations of the ASP-Core-2 \cite{te10} encodings to \textsc{FO} \cite{te12} and instances of the 2013 ASP competition \cite{te11}. We added type information (required by IDP) and constraints on the input structure if these were specified in the problem description (which were often not modeled, but are crucial for offline detection).

For the resulting encodings, we did two types of experiments \cite{te12}. In the first series of experiments, the detection and rewriting algorithm was applied to each of the encodings with one selected instance to measure how many functional dependencies were detected and how much the rewriting reduced the number of quantified variables.

\footnote{ASP-Core-2 supports no function symbols except aggregate functions.}
\footnote{See \url{https://www.mat.unical.it/aspcog2013/officialProblemSuite}}
\footnote{All experiments were run on an 64-bit Ubuntu 12.10 system with a quad-core 2.53 GHz processor and 4 Gb of RAM. All benchmarks, experimental data and results can be found on \url{http://dtai.cs.kuleuven.be/krr/files/experiments/iclp2013-function-experiments.tar.gz}}
The details of the experimental results can be found in Appendix D; here we only provide a summary. Out of 19 benchmarks, functional dependencies were detected in all but 3 benchmarks. In those 16 benchmarks, 45% of the detected dependencies were partial, of which 75% were detected in two benchmarks. The subsequent rewrite transformation erased on average 52% of all quantified variables, with peaks above 85%, and was able to strongly reduce the size of the grounding. Total detection time ranged from less than 1 second to 450 seconds and was directly proportional to the number of symbols and their arity, as most calls to SPASS timed out (a 2 second timeout was used).

Close inspection showed that the prover was unable to detect functional dependencies in constraints of the form $\forall x : P(x, t) \Leftrightarrow (x = \text{initialvalue} \land t = 0) \lor (\ldots P(y, t - 1) \ldots)$, which occur frequently when reasoning over time (e.g., in planning problems). Indeed, SPASS does not support the required inductive reasoning. While one could organize it (prove first for $t = 0$, then the induction step), our current implementation does not.

A second series of experiments was performed to evaluate the effect on the solver’s performance. These results are only preliminary, partly because at the time of writing, MINISAT(ID) only supported total functions with a numeric co-domain. The results are promising however: for each benchmark, the number of solved instances often increased significantly while the running times improved substantially for the harder problems.

To summarize, offline detection of functional dependencies is certainly worthwhile, as detected dependencies can result in a significant performance boost for the solver, while the performance is unaffected when none are detected. Whether to use online detection depends on the application at hand as the proving overhead could be significant.

6 Concluding remarks

The main contribution of this work is to use FO theorem proving to detect functional dependencies in declarative problem statements and to exploit these dependencies by rewriting the theory. This reduces the size of the grounding; moreover, the grounding can exploit the constraint programming features of the latest generation of grounders and search algorithms. Preliminary experimental results show that many functional dependencies can indeed be automatically detected and that the effect on grounding size and solver performance is often significant. Part of future work is to extract other types of implicit knowledge, such as smaller types, definitional totality and more complex finite-domain constraints such as all-different. In the field of CP, Mears et al. (2008) search for symmetries in a problem specification using a different approach, namely by generating multiple solutions for several instances, from which symmetry candidates are extracted, which can then be verified using e.g., theorem provers. Wittocx et al. (2013) have studied unit propagation on a symbolic level by deriving formulas entailed by the theory by reasoning on unit propagation schemes. It remains an open question whether theorem provers with additional reasoning capabilities (e.g. arithmetic), such as Melia (Baumgartner et al. 2012), Princess (Rümmer 2008) and Z3 (de Moura and Björner 2008) are capable of proving the presence of even more functional dependencies.

Acknowledgements Broes De Cat is funded by the Institute for Science and Technology Flanders (IWT). We thank Philipp Rümmer for sharing with us his insights into theorem provers and the reviewers for their valuable comments.
References


Litetsitzi, V. 2012. Logic programs with intensional functions. See Brewka et al. (2012).


