*TLP* **15** (4–5): 464–480, 2015. © Cambridge University Press 2015
doi:10.1017/S1471068415000162

# *Knowledge compilation of logic programs using approximation fixpoint theory*

BART BOGAERTS and GUY VAN DEN BROECK
*Department of Computer Science, KU Leuven, Belgium*
(*e-mail:* `bart.bogaerts@cs.kuleuven.be, guy.vandenbroeck@cs.kuleuven.be`)

*submitted 29 April 2015; revised 3 July 2015; accepted 15 July 2015*

**Abstract**

Recent advances in knowledge compilation introduced techniques to compile *positive* logic programs into propositional logic, essentially exploiting the constructive nature of the least fixpoint computation. This approach has several advantages over existing approaches: it maintains logical equivalence, does not require (expensive) loop-breaking preprocessing or the introduction of auxiliary variables, and significantly outperforms existing algorithms. Unfortunately, this technique is limited to *negation-free* programs. In this paper, we show how to extend it to general logic programs under the well-founded semantics.

We develop our work in approximation fixpoint theory, an algebraical framework that unifies semantics of different logics. As such, our algebraical results are also applicable to autoepistemic logic, default logic and abstract dialectical frameworks.

## 1 Introduction

There is a fundamental tension between the expressive power of a knowledge representation language, and its support for efficient reasoning. Knowledge compilation studies this tension (Cadoli and Donini 1997; Darwiche and Marquis 2002), by identifying *languages* that support certain queries and transformations efficiently. It studies the relative succinctness of these languages, and is concerned with building *compilers* that can transform knowledge bases into a desired target language. For example, after compiling two CNF sentences into the OBDD language (Bryant 1986), their equivalence can be checked in polynomial time. Applications of knowledge compilation are found in diagnosis (Huang and Darwiche 2005), databases (Suciu *et al.* 2011), planning (Palacios *et al.* 2005), graphical models (Chavira and Darwiche 2005; Fierens *et al.* 2015) and machine learning (Lowd and Domingos 2008). These techniques are most effective when the cost of compilation can be amortised over many queries to the knowledge base.

Knowledge compilation has traditionally focused on subsets of propositional logic and Boolean circuits in particular (Darwiche and Marquis 2002; Darwiche 2011). Logic programs have received much less attention, which is surprising given their historical significance in AI and current popularity in the form of answer set programming (ASP) (Marek and Truszczyński 1999). Closest in spirit are techniques to encode logic programs into CNF (Ben-Eliyahu and Dechter 1994; Lin and Zhao 2003, 2004; Janhunen 2004, 2006). A notable difference with traditional knowledge compilation is that many of these encodings are task-specific: the resulting CNF is not equivalent to the logic program. Instead, it is *equisatisfiable* for the purpose of satisfiability checking, or has an identical model count for the purpose of probabilistic inference (Fierens *et al.* 2015).[1] These encodings often introduce new variables and loop-breaking formulas, which blow up the representation. Lifschitz and Razborov (2006) showed that there can be no polynomial translation of ASP into a flat propositional logic theory without auxiliary variables.[2]

Recently, Vlasselaer *et al.* (2015) introduced a novel knowledge compilation technique for *positive* logic programs. As an example, consider the logic program $\mathscr{P}$ defining the transitive closure $r$ of a binary relation $e$:

$$\left\{ \begin{array}{ll} \forall X, Y \,: r(X,Y) & \leftarrow e(X,Y). \\ \forall X, Y, Z \,: r(X,Y) & \leftarrow e(X,Z) \wedge r(Z,Y). \end{array} \right\}$$

Intuitively, Vlasselaer *et al.* (2015) compute the minimal model of $\mathscr{P}$ for all interpretations of $e(\cdot,\cdot)$ *simultaneously*. They define a lifted least fixpoint computation where the intermediate results are symbolic interpretations of $r(\cdot,\cdot)$ in terms of $e(\cdot,\cdot)$. For example, in a domain $\{a,b,c\}$, the interpretation of $r(a,b)$ in the different steps of the least fixpoint computation would be.

$$r(a,b): \qquad \mathbf{f} \qquad \rightsquigarrow \qquad e(a,b) \qquad \rightsquigarrow \qquad e(a,b) \vee (e(a,c) \wedge e(c,b))$$

I.e., initially, $r(a,b)$ is false; next $r(a,b)$ is derived to be true if $e(a,b)$ holds; finally, $r(a,b)$ also holds if $e(a,c)$ and $e(c,b)$ hold. The result of this sequence is a symbolic, Boolean formula representation of the well-founded model for each interpretation of $e$; this formula can be used for various inference tasks. This approach has several advantages over traditional knowledge compilation methods: it preserves logical equivalence[3] (and hence, enables us to port any form of inference—e.g., abductive or inductive reasoning, (weighted) model counting, query answering, ... ) and does not require (expensive) loop-breaking preprocessing or auxiliary variables. Vlasselaer *et al.* (2015) showed that this method for compiling positive programs (into the SDD language (Darwiche 2011)) significantly outperforms traditional approaches that compile the completion of the program with added loop-breaking formulas.

Unfortunately, the methods of Vlasselaer *et al.* (2015) do not work in the presence of negation, i.e., if the immediate consequence operator is non-monotone. In this paper, we show how the well-founded model computation from Van Gelder *et al.* (1991), that works on partial interpretations, can be executed symbolically, resulting in the *parametrised well-founded model*. By doing this, we essentially compute the well-founded model of an *exponential* number of logic programs at once.

[1] Probabilistic inference on the CNF may itself perform a second knowledge compilation step.

[2] Similar, task-specific, translation techniques of logic programs into difference logic (Janhunen *et al.* 2009) and ordered completion (Asuncion *et al.* 2012) exist.

[3] In the sense that an interpretation is a model of the resulting propositional theory if and only if it is a model of the given logic program under the parametrised well-founded semantics.

Our algorithm works in principle on any representation of Boolean formulas; we study complexity for this algorithm taking Boolean circuits as target language; in this case we find that our algorithm has *polynomial* time complexity. General Boolean circuits are not considered to be an interesting target language, as they are not tractable for any query of interest. However, what we achieve here is a *change of semantic paradigm* that uncovers all the machinery for propositional logic (SAT solvers, model counters, etc.). It is a required step before further compiling the circuit into a language such as OBDD or SDD, which do permit tractable querying. It is also possible to encode the circuit into CNF, similar to Janhunen (2004). There is a long list of queries and transformations that become supported on logic programs (under the well-founded semantics), by virtue of our algorithm. After a transformation to propositional logic, we can use standard tools to check whether one logic program is entailed by another, find models that are minimal with respect to some optimisation term, check satisfiability, count or enumerate models, and forget or condition variables (Darwiche and Marquis 2002). For example, the following definition of the transitive closure of $e$ syntactically differs from the previous.

$$\left\{\begin{array}{ll} \forall X, Y \ : r(X, Y) & \leftarrow e(X, Y). \\ \forall X, Y, Z \ : r(X, Y) & \leftarrow r(X, Z) \wedge r(Z, Y). \end{array}\right\}$$

With our algorithm, we can compile both programs into an OBDD representation. On these OBDDs, we can verify the equivalence of the logic programs using existing OBDD algorithms. As logic programs under the well-founded semantics encode *inductive definitions* (Denecker and Vennekens 2014), we now have the machinery to check that two definitions define the same concept for each interpretation of the parameters ($e$ in our example). Moreover, our algorithm can be stopped at any time to obtain upper and lower bounds on the fixpoint, which gives us *approximate knowledge compilation* for logic programs (Selman and Kautz 1996).

The original motivation for this research is the fact that probabilistic inference tools such as ProbLog (Fierens *et al.* 2015) use knowledge compilation for probabilistic inference by (weighted) model counting; they compile a logic program into a d-DNNF or SDD (with auxiliary variables) and subsequently calling a weighted model counter. Vlasselaer *et al.* showed that for *positive* logic programs, this can be done much more efficiently using bottom-up compilation techniques. We extend these techniques to general logic programs to capture the full ProbLog language.

More generally, we develop our ideas in *approximation fixpoint theory* (AFT), an abstract algebraical theory that captures all common semantics of logic programming, autoepistemic logic, default logic, Dung's argumentation frameworks and abstract dialectical frameworks (as shown by Denecker *et al.* (2000) and Strass (2013)). Afterwards, we show how the algebraical results apply to logic programming. We thus extend the ideas by Vlasselaer *et al.* (2015) in two ways; first, by developing a theory that works for *general* logic programs and secondly by lifting the theory to the algebraical level. Due to the high level of abstraction, our proofs are (relatively) compact and our algebraical results are immediately applicable to all aforementioned paradigms. Due to page restrictions, proofs are postponed to the online appendix (Appendix B) and we only apply our theory to logic programming.

Summarised, the main contributions of this paper are as follows: *(i)* we present the algebraical foundations for a novel knowledge compilation technique for *general* logic programs, *(ii)* we apply the algebraical theory to logic programming, resulting in a family of equivalence-preserving algorithms, *(iii)* we show that Boolean circuits are at least as succinct as propositional logic programs (under the parametrised well-founded semantics), and *(iv)* we pave the way towards knowledge compilation for other non-monotonic formalisms, such as autoepistemic logic.

## 2 Preliminaries

### *2.1 Lattices and approximation fixpoint theory*

A *complete lattice* $\langle L, \leqslant \rangle$ is a set $L$ equipped with a partial order $\leqslant$ such that every subset $S$ of $L$ has a *least upper bound*, denoted $\bigvee S$ and a *greatest lower bound*, denoted $\bigwedge S$. If $x$ and $y$ are two lattice elements, we use the notations $x \wedge y = \bigwedge\{x, y\}$ and $x \vee y = \bigvee\{x, y\}$. A complete lattice has a least element $\bot$ and a greatest element $\top$. An operator $O : L \to L$ is *monotone* if $x \leqslant y$ implies that $O(x) \leqslant O(y)$. Every monotone operator $O$ in a complete lattice has a least fixpoint, denoted $\mathrm{lfp}(O)$. A mapping $f : (L, \leqslant_L) \to (K, \leqslant_K)$ between lattices is a *lattice morphism* if it preserves least upper bounds and greatest lower bounds, i.e. if for every subset $X$ of $L$, $f(\bigvee X) = \bigvee f(X)$ and $f(\bigwedge X) = \bigwedge f(X)$.

Given a lattice, approximation fixpoint theory makes uses of the bilattice $L^2$. We define *projections* as usual: $(x, y)_1 = x$ and $(x, y)_2 = y$. Pairs $(x, y) \in L^2$ are used to approximate all elements in the interval $[x, y] = \{z \mid x \leqslant z \wedge z \leqslant y\}$. We call $(x, y) \in L^2$ *consistent* if $x \leqslant y$, that is, if $[x, y]$ is non-empty. We use $L^c$ to denote the set of consistent pairs. Pairs $(x, x)$ are called *exact*. The *precision ordering* on $L^2$ is defined as $(x, y) \leqslant_p (u, v)$ if $x \leqslant u$ and $v \leqslant y$. In case $(u, v)$ is consistent, $(x, y)$ is less precise than $(u, v)$ if $(x, y)$ approximates all elements approximated by $(u, v)$, or in other words if $[u, v] \subseteq [x, y]$. If $L$ is a complete lattice, then so is $\langle L^2, \leqslant_p \rangle$.

AFT studies fixpoints of operators $O : L \to L$ through operators approximating $O$. An operator $A : L^2 \to L^2$ is an *approximator* of $O$ if it is $\leqslant_p$-monotone, and has the property that for all $x$, $O(x) \in A(x, x)$. Approximators are internal in $L^c$ (i.e., map $L^c$ into $L^c$). As usual, we restrict our attention to *symmetric* approximators: approximators $A$ such that for all $x$ and $y$, $A(x, y)_1 = A(y, x)_2$. Denecker *et al.* (2004) showed that the consistent fixpoints of interest are uniquely determined by an approximator's restriction to $L^c$, hence, we only define approximators on $L^c$.

AFT studies fixpoints of $O$ using fixpoints of $A$. The $A$-Kripke-Kleene fixpoint is the $\leqslant_p$-least fixpoint of $A$ and has the property that it approximates all fixpoints of $O$. A partial $A$-stable fixpoint is a pair $(x, y)$ such that $x = \mathrm{lfp}(A(\cdot, y)_1)$ and $y = \mathrm{lfp}(A(x, \cdot)_2)$. The $A$-well-founded fixpoint is the least precise partial $A$-stable fixpoint. An $A$-*stable fixpoint* of $O$ is a fixpoint $x$ of $O$ such that $(x, x)$ is a partial $A$-stable fixpoint. The $A$-Kripke-Kleene fixpoint of $O$ can be constructed by iteratively applying $A$, starting from $(\bot, \top)$. For the $A$-well-founded fixpoint, Denecker and Vennekens (2007) worked out a similar constructive characterisation as follows.

An $A$-*refinement* of $(x, y)$ is a pair $(x', y') \in L^2$ satisfying one of the following conditions *(i)* $(x, y) \leqslant_p (x', y') \leqslant_p A(x, y)$, or *(ii)* $x' = x$ and $A(x, y')_2 \leqslant y' \leqslant y$.

An $A$-refinement is *strict* if $(x, y) \neq (x', y')$. We call refinements of the first kind *application refinements* and refinements of the second kind *unfoundedness refinements.* A *well-founded induction* of $A$ is a sequence $(x_i, y_i)_{i \leqslant \beta}$ with $\beta$ an ordinal such that

- $(x_0, y_0) = (\bot, \top)$;
- $(x_{i+1}, y_{i+1})$ is an A-refinement of $(x_i, y_i)$, for all $i < \beta$;
- $(x_\lambda, y_\lambda) = \bigvee_{\leqslant_p} \{(x_i, y_i) \mid i < \lambda\}$ for each limit ordinal $\lambda \leqslant \beta$.

A well-founded induction is *terminal* if its limit $(x_\beta, y_\beta)$ has no strict $A$-refinements. For a given approximator $A$, there are many different terminal well-founded inductions of $A$. Denecker and Vennekens (2007) showed that they all have the same limit, which equals the $A$-well-founded fixpoint of $O$. Denecker and Vennekens (2007) also showed how to obtain maximally precise unfoundedness refinements.

*Proposition 2.1* (*Denecker and Vennekens, 2007*)
Let $A$ be an approximator of $O$ and $(x, y) \in L^2$. Let $S_A^x$ be the operator on $L$ that maps every $y'$ to $A(x, y')_2$. This operator is monotone. The smallest $y'$ such that $(x, y')$ is an unfoundedness refinement of $(x, y)$ is given by $y' = \mathrm{lfp}(S_A^x)$.

## 2.2 *Logic programming*

In this paper, we restrict our attention to propositional logic programs. However, AFT has been applied in a much broader context (Denecker *et al.* 2000; Pelov *et al.* 2007; Antic *et al.* 2013) and our results apply in these richer settings as well.

Let $\Sigma$ be an alphabet, i.e., a collection of symbols called *atoms.* A *literal* is an atom $p$ or its negation $\neg p$. A logic program $\mathscr{P}$ is a set of *rules* $r$ of the form $h \leftarrow l_1 \wedge l_2 \wedge \cdots \wedge l_n$, where $h$ is an atom called the *head* of $r$, denoted *head*$(r)$, and the $l_i$ are literals. The formula $l_1 \wedge l_2 \wedge \cdots \wedge l_n$ is the *body* of $r$, denoted *body*$(r)$. A rule $r = \forall \overline{X} : h \leftarrow \varphi$ is, as usual, a shorthand for the *grounding of* $r$, the collection of rules obtained by substituting the variables $\overline{X}$ by elements from a given domain. If $p \in \Sigma$, the formula $\varphi_p$ is $\bigvee_{r \in \mathscr{P} \wedge head(r) = p} body(r)$. An interpretation $I$ of the alphabet $\Sigma$ is an element of $2^\Sigma$, i.e., a subset of $\Sigma$. The set of interpretations $2^\Sigma$ forms a lattice equipped with the order $\subseteq$. The truth value ($\mathbf{t}$ or $\mathbf{f}$) of a propositional formula $\varphi$ in a structure $I$, denoted $\varphi^I$ is defined as usual. With a logic program $\mathscr{P}$, we associate an immediate consequence operator (van Emden and Kowalski 1976) $T_\mathscr{P}$ mapping structure $I$ to $T_\mathscr{P}(I) = \{p \mid \varphi_p^I = \mathbf{t}\}$.

In the context of logic programming, elements of the bilattice $\left(2^\Sigma\right)^2$ are four-valued interpretations, pairs $\mathscr{I} = (I_1, I_2)$ of interpretations. A four-valued interpretation maps atoms $p \in \Sigma$ to tuples of two truth values $(p^{I_1}, p^{I_2})$. Such tuples are often identified with four-valued truth values (true ($\mathbf{t}$), false ($\mathbf{f}$), unknown ($\mathbf{u}$) and inconsistent ($\mathbf{i}$)). Intuitively, $p^{I_1}$ represents whether $p$ is true, and $p^{I_2}$ whether $p$ is possible, i.e., not false. Thus, the following correspondence holds $\mathbf{t} = (\mathbf{t}, \mathbf{t}), \mathbf{f} = (\mathbf{f}, \mathbf{f}), \mathbf{u} = (\mathbf{f}, \mathbf{t})$ (and $\mathbf{i} = (\mathbf{t}, \mathbf{f})$). The pair $(I_1, I_2)$ approximates all interpretations $I'$ with $I_1 \subseteq I' \subseteq I_2$. We are mostly concerned with consistent (also called partial) interpretations: tuples $(I_1, I_2)$ with $I_1 \subseteq I_2$, i.e., interpretations that map no atoms to $\mathbf{i}$. If $\mathscr{I}$ is a partial interpretation, and $\varphi$ a formula, we write $\varphi^\mathscr{I}$ for the standard three-valued valuation based on Kleene's truth tables (Kleene 1938). We often identify interpretation $I$ with the partial interpretation $(I, I)$.

The most common approximator for logic programs is Fitting's (2002) immediate consequence operator $\Psi_{\mathscr{P}}$ , a generalisation of $T_{\mathscr{P}}$ to partial interpretations:

$$\Psi_{\mathscr{P}}(\mathscr{I})_1 = \{a \in \Sigma \mid \exists r \in \mathscr{P} : body(r)^{\mathscr{I}} = \mathbf{t} \wedge head(r) = a\},$$
$$\Psi_{\mathscr{P}}(\mathscr{I})_2 = \{a \in \Sigma \mid \exists r \in \mathscr{P} : body(r)^{\mathscr{I}} \neq \mathbf{f} \wedge head(r) = a\}$$

Denecker *et al.* (2000) showed that the $\Psi_{\mathscr{P}}$-well-founded fixpoint of $T_{\mathscr{P}}$ is the well-founded model of $\mathscr{P}$ (Van Gelder *et al.* 1991) and that $\Psi_{\mathscr{P}}$-stable fixpoints are exactly the stable models of $\mathscr{P}$ (Gelfond and Lifschitz 1988).

*Parametrised Logic Programs* We briefly recall the parametrised well-founded semantics. This semantics has been implicitly present in the literature for a long time, by assigning a meaning to an *intensional* database. We follow the formalisation by Denecker and Vennekens (2007). For *parametrised logic programs*, the alphabet $\Sigma$ is partitioned into a set $\Sigma_p$ of parameter symbols and a set $\Sigma_d$ of defined symbols. Only defined symbols occur in heads of rules. Given a $\Sigma_p$-interpretation $I$, $\mathscr{P}$ defines an immediate consequence operator $T^I_{\mathscr{P}} : 2^{\Sigma_d} \to 2^{\Sigma_d}$ equal to $T_{\mathscr{P}}$ except that the value of atoms in $\Sigma_p$ is fixed to their value in $I$. Similarly, Fitting's immediate consequence operator $\Psi^I_{\mathscr{P}}$ induces an operator on $(2^{\Sigma_d})^2$. $J$ is a *model*[4] of $\mathscr{P}$ under the parametrised well-founded semantics (denoted $J \models_{wf} \mathscr{P}$) if $J \cap \Sigma_d$ is the $\Psi^{J\cap\Sigma_p}_{\mathscr{P}}$-well-founded fixpoint of $T^{J\cap\Sigma_p}_{\mathscr{P}}$. By adding a probability distribution over the parameter symbols, we obtain the ProbLog language (Fierens *et al.* 2015).

## 3 Algebraical theory

In this section we develop the algebraical foundations of our techniques. We follow the intuitions presented in the introduction: we define one operator that "summarises" an entire family operators (these will be immediate consequence operators for different interpretations of the parameter symbols). We study the relationship between the well-founded fixpoint of the summarising operator and the original operators. Before formally introducing *parametrisations*, we focus on a simpler situation: we show that surjective lattice morphisms preserve the well-founded fixpoint.

### *3.1 Surjective lattice morphisms*

*Definition-Proposition 3.1*
Let $O : L \to L$ be an operator and $f : L \to K$ a lattice morphism. We say that $O$ *respects* $f$ if for every $x, y \in L$ with $f(x) = f(y)$, it holds that $f(O(x)) = f(O(y))$.

If $f$ is surjective and $O$ respects $f$, then there exists a unique operator $O_f : K \to K$ with $O_f \circ f = f \circ O$, which we call the *projection of $O$ on $K$*.

[4] Note that this definition of model differs from the traditional definition of model of a logic program. To emphasise this difference, we use $J \models_{wf} \mathscr{P}$ to refer to the parametrised well-founded semantics and $J \models \mathscr{T}$ for the satisfaction relation of propositional logic.

$$\begin{tikzcd} \langle L,\leq\rangle \arrow[r, two heads, "f"] \arrow[d, squiggly] \arrow[loop left, "O"] & \langle K,\leq\rangle \arrow[d, squiggly] \arrow[loop right, "O_f"] \\ \langle L^2,\leq_p\rangle \arrow[r, two heads, "f^2"] \arrow[loop left, "A"] & \langle K^2,\leq_p\rangle \arrow[loop right, "A_f"] \end{tikzcd}$$

Fig. 1. Overview of the operators

If $f : L \to K$ is a lattice morphism, $f^2 : L^2 \to K^2 : (x, y) \mapsto (f(x), f(y))$ is a lattice morphism from the bilattice $L^2$ to the bilattice $K^2$.

*Definition 3.2*
Let $A : L^2 \to L^2$ be an approximator and $f : L \to K$ a lattice morphism. We say that $A$ *respects* $f$ if $A$ respects $f^2$ in the sense of Definition 3.1. Furthermore, if $f$ is surjective, we define the *projection* of $A$ on $K$ as the unique operator $A_f : K^2 \to K^2$ with $A_f \circ f^2 = f^2 \circ A$.

Below, we assume that $f : L \to K$ is a surjective lattice morphism, that $O : L \to L$ is an operator and $A : L^2 \to L^2$ an approximator of $O$ such that both $O$ and $A$ respect $f$ (see Figure 1). Intuitively elements of $L$ can be thought of as symbolic representations of interpretations, while the elements of $K$ are classical interpretations.

The following proposition explicates the relationship between well-founded inductions in $L$ and in $K$. This proposition immediately leads to a relationship between the $A$-well-founded model of $O$ and the $A_f$-well-founded model of $O_f$.

*Proposition 3.3*
If $(x_j, y_j)_{j\leqslant\alpha}$ is a well-founded induction of $A$, then $(f(x_j), f(y_j))_{j\leqslant\alpha}$ is a well-founded induction of $A_f$. If $(x_j, y_j)_{j\leqslant\alpha}$ is terminal, then so is $(f(x_j), f(y_j))_{j\leqslant\alpha}$.

*Theorem 3.4*
If $(x, y)$ is the $A$-well-founded fixpoint of $O$, then, $(f(x), f(y))$ is the $A_f$-well-founded fixpoint of $O_f$.

## ***3.2 Parametrisations***

*Definition 3.5*
Let $L$ and $K$ be lattices. Suppose $(f_i : L \to K)_{i\in I}$ is a family of surjective lattice morphisms. We call $L$ a *parametrisation* of $K$ (through $(f_i)_{i\in I}$) if for every $x, y \in L$ it holds that $x \leqslant y$ if and only if for every $i \in I$, $f_i(x) \leqslant f_i(y)$.

A parametrisation $L$ of a lattice $K$ can be used to "summarise" multiple operators (the $O_{f_i}$) on $K$ by means of a single operator $O$ on $L$ which abstracts away certain details. In the next section, we use this to compute a symbolic representation of the parametrised well-founded model.

*Theorem 3.6*
Suppose $L$ is a parametrisation of $K$ through $(f_i)_{i\in I}$. Let $O : L \to L$ be an operator and $A$ an approximator of $O$ such that both $O$ and $A$ respect each of the $f_i$. If $(x, y)$ is the $A$-well-founded fixpoint of $O$, the following hold.

1. For each $i$, $(f_i(x), f_i(y))$ is the $A_{f_i}$-well-founded fixpoint of $O_{f_i}$.
2. If the $A_{f_i}$-well-founded fixpoint of $O_{f_i}$ is exact for every $i$, then so is the $A$-well-founded fixpoint of $O$.

## 4 Operator-based knowledge compilation

We assume throughout this section that $\mathscr{P}$ refers to a parametrised logic program with parameters $\Sigma_p$ and defined symbols $\Sigma_d$. In order to apply our theory to logic programming, we will define an operator (and approximator) that summarises the immediate consequence operators of $\mathscr{P}$ for all $\Sigma_p$-interpretations.

Partial interpretations map defined atoms to a tuple $(t, p)$ of two-valued truth values. We generalise this type of interpretations: we want (partial) interpretations to be parametrised in terms of the parameters of the logic program. Instead of assigning a tuple $(t, p)$ of Boolean values to each atom, we will hence assign a tuple of two propositional formulas over $\Sigma_p$ to each atom in $\Sigma_d$.

In order to avoid redundancies, we work *modulo equivalence*. Let $\mathscr{L}_{\Sigma_p}$ be the language of all propositional formulas over vocabulary $\Sigma_p$. If $\varphi$ is a propositional formula, we use $\bar{\varphi}$ to denote the equivalence class of $\varphi$, i.e., the set of propositional formulas equivalent to $\varphi$.[5] Let $L_p$ be the set of equivalence classes of elements in $\mathscr{L}_{\Sigma_p}$. We define an order $\leqslant_{L_p}$ on $L_p$ as follows: $\bar{\varphi} \leqslant_{L_p} \bar{\psi}$ if $\varphi$ entails $\psi$ (in standard propositional logic). This order is well-defined (independent of the choice of representatives $\varphi$ and $\psi$); with this order, $L_p$ is a complete lattice. Boolean operations on $L_p$ are defined by applying them to representatives.

*Definition 4.1*
A *symbolic interpretation* of $\Sigma_d$ in terms of $\Sigma_p$ is a mapping $\Sigma_d \to L_p$. The *symbolic interpretation lattice* $L_p^d$ is the set of all symbolic interpretations of $\Sigma_d$ in terms of $\Sigma_p$. The order $\leqslant$ on $L_p^d$ is the pointwise extension of $\leqslant_{L_p}$. A *partial symbolic interpretation* is an element of the bilattice $(t, p) \in (L_p^d)^2$ such that $t \leqslant p$.

The condition $t \leqslant p$ in Definition 4.1 excludes inconsistent interpretations. If $\Sigma_p$ is the empty vocabulary (i.e., if $\mathscr{P}$ has no parameters), then the lattice $L_p$ is $\{\bar{\mathbf{f}}, \bar{\mathbf{t}}\}$ with order $\bar{\mathbf{f}} \leqslant \bar{\mathbf{t}}$. Hence, in this case, a (partial) symbolic interpretation is "just" a (partial) interpretation. As with classical interpretations, we often identify a symbolic interpretation $\mathscr{A}$ with the partial symbolic interpretation $(\mathscr{A}, \mathscr{A})$.

Intuitively, a (partial) symbolic interpretation summarises many different classical (partial) interpretations; when we instantiate such as (partial) symbolic interpretation with a $\Sigma_p$-interpretation, we obtain a unique (partial) $\Sigma_d$-interpretation. The following definition formalises this intuition.

*Definition 4.2*
If $\mathscr{S} = (\mathscr{A}_t, \mathscr{A}_p)$ is a partial symbolic interpretation and $I$ is a $\Sigma_p$-interpretation, the *concretisation* of $\mathscr{S}$ by $I$ is the partial interpretation $\mathscr{S}^I$ such that for every symbol $a \in \Sigma_d$ with $\mathscr{A}_t(a) = \overline{\varphi_t}$ and $\mathscr{A}_p(a) = \overline{\varphi_p}$, it holds that $\mathscr{S}^I(a) = (\varphi_t^I, \varphi_p^I)$.

[5] Notice that $\bar{a}$ is *not* the negation of an atom $a$. We use $\neg a$ for the negation of $a$.

The above concept is well-defined (independent of the choice of representatives $\varphi_t$ en $\varphi_p$). A symbolic interpretation can thus be seen as a mapping from $\Sigma_p$-interpretations to $\Sigma_d$-interpretations. This kind of mapping is of particular interest, since the parametrised well-founded semantics induces a similar mapping: it associates with every $\Sigma_p$-interpretation a $\Sigma_d$-interpretation, namely the $\Psi^I_{\mathscr{P}}$-well-founded model of $T^I_{\mathscr{P}}$. It is this relationship between $\Sigma_p$- and $\Sigma_d$-interpretations that we wish to capture in propositional logic. Furthermore, as explained below, it is easy to translate a symbolic interpretation into propositional logic.

*Definition 4.3*
Let $\mathscr{A}$ be a symbolic interpretation and $\psi_p$ a representative of $\mathscr{A}(p)$ for each $p \in \Sigma_d$. We call a propositional theory $\mathscr{T}$ *a theory of* $\mathscr{A}$ if it is equivalent to $\bigwedge_{p\in\Sigma_d} p \Leftrightarrow \psi_p$.

All theories of $\mathscr{A}$ are equivalent. We sometimes abuse notation and refer to *the* theory of $\mathscr{A}$, denoted $Th(\mathscr{A})$, to refer to any theory from this class. The goal now is to find a symbolic interpretation $\mathscr{A}$ such that $Th(\mathscr{A})$ is equivalent to $\mathscr{P}$. Our choice of representatives will depend on the target language of the compilation.

The value of a propositional formula $\varphi$ in a partial interpretation $\mathscr{I}$ is an element of $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ (or, a tuple of two Booleans) obtained by standard three-valued valuation. This can easily be extended to symbolic interpretations, where the value of a formula in a (partial) symbolic interpretation is a tuple of two $\Sigma_p$ formulas.

*Definition 4.4*
Let $\varphi$ be a $\Sigma$-formula and $\mathscr{S} = (\mathscr{A}_t, \mathscr{A}_p)$ a partial symbolic interpretation. The value of $\varphi$ in $\mathscr{S}$ is a tuple $(\varphi_t, \varphi_p) \in L_p^2$ defined inductively as follows:

- $p^{(\mathscr{A}_t,\mathscr{A}_p)} = (\bar{p}, \bar{p})$ if $p \in \Sigma_p$ and $p^{(\mathscr{A}_t,\mathscr{A}_p)} = (\mathscr{A}_t(p), \mathscr{A}_p(p))$ if $p \in \Sigma_d$,
- $(\psi \wedge \xi)^{(\mathscr{A}_t,\mathscr{A}_p)} = (\overline{\psi_t \wedge \xi_t}, \overline{\psi_p \wedge \xi_p})$ if $\psi^{(\mathscr{A}_t,\mathscr{A}_p)} = (\overline{\psi_t}, \overline{\psi_p})$ and $\xi^{(\mathscr{A}_t,\mathscr{A}_p)} = (\overline{\xi_t}, \overline{\xi_p})$
- $(\psi \vee \xi)^{(\mathscr{A}_t,\mathscr{A}_p)} = (\overline{\psi_t \vee \xi_t}, \overline{\psi_p \vee \xi_p})$ if $\psi^{(\mathscr{A}_t,\mathscr{A}_p)} = (\overline{\psi_t}, \overline{\psi_p})$ and $\xi^{(\mathscr{A}_t,\mathscr{A}_p)} = (\overline{\xi_t}, \overline{\xi_p})$
- $(\neg\psi)^{(\mathscr{A}_t,\mathscr{A}_p)} = (\overline{\neg\psi_p}, \overline{\neg\psi_t})$ if $\psi^{(\mathscr{A}_t,\mathscr{A}_p)} = (\overline{\psi_t}, \overline{\psi_p})$.

Evaluation of formulas has some nice properties. It commutes with concretisation (Proposition 4.5) and induces a parametrisation (Proposition 4.6).

*Proposition 4.5*
For every formula $\varphi$ over $\Sigma$, $\mathscr{S} \in (L_p^d)^2$ and $I \in 2^{\Sigma_p}$, it holds that $\varphi^{\mathscr{S}^I} = (\varphi^{\mathscr{S}})^I$.

*Proposition 4.6*
The lattice $L_p^d$ is a parametrisation of $2^{\Sigma_d}$ through the mappings $(\pi_I : L_p^d \rightarrow 2^{\Sigma_d} : \mathscr{A} \mapsto \mathscr{A}^I)_{I\in 2^{\Sigma_p}}$.

Recall from Section 2.2 that $\varphi_p$ is the disjunction of all bodies of rules defining $p$; using this we can generalise both $T_{\mathscr{P}}$ and $\Psi_{\mathscr{P}}$ to a symbolic setting.

*Definition 4.7*
The *partial parametrised immediate consequence operator* $\mathit{\Psi}_{\mathscr{P}} : (L_p^d)^2 \rightarrow (L_p^d)^2$ is defined by $\mathit{\Psi}_{\mathscr{P}}(\mathscr{S})(p) = \varphi_p^{\mathscr{S}}$ for every $p \in \Sigma_d$.

The *parametrised immediate consequence operator* is the operator $\mathscr{T}_{\mathscr{P}} : L_p^d \rightarrow L_p^d$ that maps $\mathscr{A}$ to $\mathscr{T}_{\mathscr{P}}(\mathscr{A})$, where $\mathscr{T}_{\mathscr{P}}(\mathscr{A})(p) = \varphi_p^{\mathscr{A}}$ for each $p \in \Sigma_d$.

It deserves to be noticed that the operator $\mathcal{T}_{\mathcal{P}}$ almost coincides with the operator $\mathcal{T}_{c_{\mathcal{P}}}$ defined by Vlasselaer *et al.* (2015) (the only difference is that we work modulo equivalence). The following proposition, which follows easily from our algebraical theory, shows correctness of the methods developed by Vlasselaer *et al.* (2015).

*Theorem 4.8*
If $\mathcal{P}$ is a positive logic program, then $\mathcal{T}_{\mathcal{P}}$ is monotone. For every $\Sigma$-interpretation $I$, it then holds that $I \models_{wf} \mathcal{P}$ if and only if $I \models Th(\mathrm{lfp}(\mathcal{T}_{\mathcal{P}}))$.

*Theorem 4.9*
For any parametrised logic program $\mathcal{P}$, the following hold:

1. $\Psi_{\mathcal{P}}$ is an approximator of $\mathcal{T}_{\mathcal{P}}$.
2. For every $\Sigma_p$-structure $I$, it holds that $\Psi^I_{\mathcal{P}} \circ \pi^2_I = \pi^2_I \circ \Psi_{\mathcal{P}}$.

*Definition 4.10*
Let $\mathcal{P}$ be any parametrised logic program. The *parametrised well-founded model* of $\mathcal{P}$ is the $\Psi_{\mathcal{P}}$-well-founded fixpoint of $\mathcal{T}_{\mathcal{P}}$.

Applying Theorem 3.4, combined with Proposition 4.5 and Theorem 4.9 yields:

*Theorem 4.11*
If the parametrised well-founded model of $\mathcal{P}$ is exact, i.e., of the form $(\mathcal{A}, \mathcal{A})$ for some symbolic interpretation $\mathcal{A}$, then for every $\Sigma$-interpretation $I$, it holds that $I \models_{wf} \mathcal{P}$ if and only if $I \models Th(\mathcal{A})$.

*Example 4.12*
We illustrate the various concepts introduced above on the smokers problem, a popular problem in probabilistic logic programming. Consider a group of people. A person of this group smokes if he is stressed, or if he is friends with a smoker. This results in the following logic program $\mathcal{P}_s$ with a domain of three people $\{a, b, c\}$:

$$\left\{ \begin{array}{ll} \forall X & : smokes(X) \leftarrow stress(X) \\ \forall X, Y & : smokes(X) \leftarrow fr(X, Y) \wedge smokes(Y) \end{array} \right\}$$

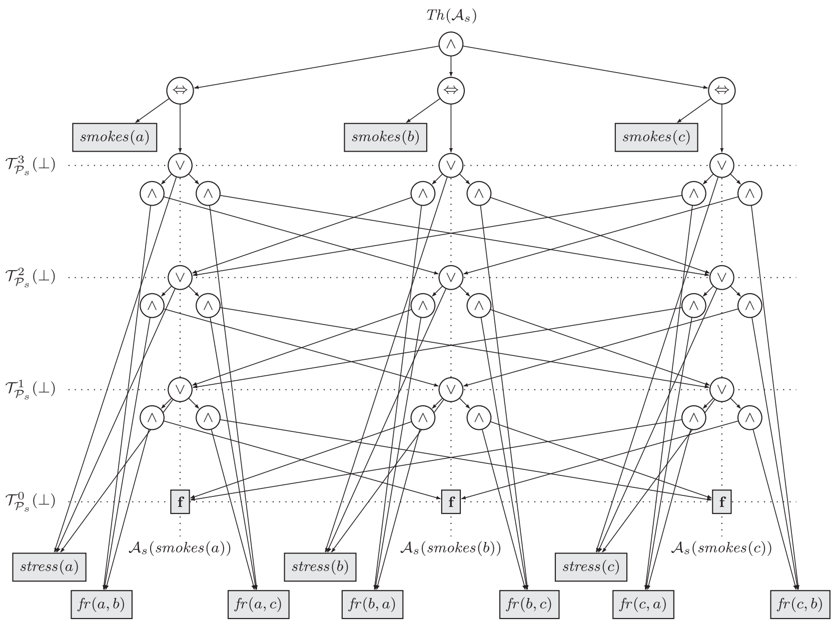
This program has parameters $stress(\cdot)$ and $fr(\cdot, \cdot)$ and defined symbols $smokes(\cdot)$. The parametrised well-founded model of $\mathcal{P}_s$ is the symbolic interpretation $\mathcal{A}_s : \Sigma_d \to L_p$ : such that

$$\begin{aligned} \mathcal{A}_s(smokes(a)) = & \overline{stress(a) \vee (stress(b) \wedge fr(a, b)) \vee (stress(c) \wedge fr(a, c))} \\ & \overline{\vee (stress(c) \wedge fr(b, c) \wedge fr(a, b))} \\ & \overline{\vee (stress(b) \wedge fr(c, b) \wedge fr(a, c))} \end{aligned}$$

and symmetrical equations hold for $smokes(b)$ and $smokes(c)$.

Notice that $Th(\mathcal{A}_s)$ is equivalent to $\mathcal{P}_s$, in the sense that $J \models Th(\mathcal{A}_s)$ if and only if $J \models_{wf} \mathcal{P}_s$. For example, let $I$ be the $\Sigma_p$-interpretation $\{stress(a), fr(b, a)\}$. We know that the $\Psi^I_{\mathcal{P}_s}$-well-founded fixpoint of $T^I_{\mathcal{P}_r}$ is $I' := \{smokes(a), smokes(b)\}$; this equals $\mathcal{A}^I_s$ and $I \cup I'$ is indeed a model of $Th(\mathcal{A}_s)$.

Since $\mathcal{P}_s$ is positive, $\mathcal{T}_{\mathcal{P}_s}$ is monotone and its least fixpoint can be computed by iteratively applying the operator $\mathcal{T}_{\mathcal{P}_s}$ starting from the smallest symbolic



$Th(\mathcal{A}_s)$
$smokes(a)$
$smokes(b)$
$smokes(c)$
$\mathcal{T}^3_{\mathcal{P}_s}(\bot)$
$\mathcal{T}^2_{\mathcal{P}_s}(\bot)$
$\mathcal{T}^1_{\mathcal{P}_s}(\bot)$
$\mathcal{T}^0_{\mathcal{P}_s}(\bot)$
f
f
f
$\mathcal{A}_s(smokes(a))$
$\mathcal{A}_s(smokes(b))$
$\mathcal{A}_s(smokes(c))$
$stress(a)$
$stress(b)$
$stress(c)$
$fr(a,b)$
$fr(a,c)$
$fr(b,a)$
$fr(b,c)$
$fr(c,a)$
$fr(c,b)$

Fig. 2. A circuit representation of the smokers theory $Th(\mathscr{A}_s)$ and the different steps in the computation of $\mathscr{T}_{\mathscr{P}_s}$.

interpretation; this yields the following sequence (only the value of *smokes*$(a)$ is explicated; for *smokes*$(b)$ and *smokes*$(c)$, similar equations hold):

$$\begin{aligned}
\bot : &\quad smokes(a) \mapsto \bar{\mathbf{f}}\\
\mathscr{T}_{\mathscr{P}_s}(\bot) : &\quad smokes(a) \mapsto \overline{stress(a)}\\
\mathscr{T}^2_{\mathscr{P}_s}(\bot) : &\quad smokes(a) \mapsto \overline{stress(a) \vee (stress(b) \wedge fr(a,b)) \vee (stress(c) \wedge fr(a,c))}\\
\mathscr{T}^3_{\mathscr{P}_s}(\bot) = &\quad \mathscr{A}_s.
\end{aligned}$$

In Figure 2, a circuit representation of $Th(\mathscr{A}_s)$ is depicted. In this circuit, the different layers correspond to different steps in the computation of the parametrised well-founded model of $\mathscr{P}_s$. Figure 2 essentially contains proofs of atoms *smokes*$(\cdot)$; this illustrates that the compiled theory can be used for example for abduction.

For general logic programs, $\mathscr{T}_{\mathscr{P}}$ is not guaranteed to be monotone and hence the parametrised well-founded model cannot be computed by iteratively applying $\mathscr{T}_{\mathscr{P}}$. Luckily, well-founded inductions provide us with a constructive way to compute it.

*Example 4.13*
Consider a dynamic domain in which two gear wheels are connected. Both wheels can be activated by an external force; since they are connected, whenever one wheel turns, so does the other. Both wheels are connected to a button. If an operator hits the button associated to some gear wheel, this means that he intends the state of the wheel to change (if a wheel was turning, its external force is turned off, if the wheel was standing still, its external force is activated). If the operator does not hit the button, the external force is set to the current state of the wheel. Initially, both

external forces are inactive. This situation (limited to two time points) is modelled in the following logic program $\mathscr{P}_w$ ($turns_i(T)$ means that wheel $i$ is turning at time point $T$ and $button_i(T)$ means that the button of wheel $i$ is pressed at time $T$):

$$\left\{ \begin{array}{ll} turns_1(0) \leftarrow turns_2(0) & turns_2(0) \leftarrow turns_1(0) \\ turns_1(1) \leftarrow turns_2(1) & turns_2(1) \leftarrow turns_1(1) \\ turns_1(1) \leftarrow turns_1(0) \wedge \neg button_1(0) & turns_2(1) \leftarrow turns_2(0) \wedge \neg button_2(0) \\ turns_1(1) \leftarrow \neg turns_1(0) \wedge button_1(0) & turns_2(1) \leftarrow \neg turns_2(0) \wedge button_2(0) \end{array} \right\}$$

This logic program has defined symbols $turns_\cdot(\cdot)$ and parameters $button_\cdot(\cdot)$. The parametrised well-founded model of $\mathscr{P}_w$ is computed by a well-founded induction of $\Psi_{\mathscr{P}_w}$. We start from the least precise partial symbolic interpretation, i.e., $\mathscr{S}_0$ that maps every $turns_\cdot(\cdot)$ to $(\bar{\mathbf{f}}, \bar{\mathbf{t}})$. Since $\mathscr{S}_0$ is a fixpoint of $\Psi_{\mathscr{P}_w}$, the only possible type of refinement is unfoundedness refinement, resulting in $\mathscr{S}_1$ that maps

$$\begin{array}{ll} turns_1(0) \mapsto (\bar{\mathbf{f}}, \bar{\mathbf{f}}) & turns_2(0) \mapsto (\bar{\mathbf{f}}, \bar{\mathbf{f}}) \\ turns_1(1) \mapsto (\bar{\mathbf{f}}, \bar{\mathbf{t}}) & turns_2(1) \mapsto (\bar{\mathbf{f}}, \bar{\mathbf{t}}) \end{array}$$

Application refinement then results in the partial symbolic interpretation $\mathscr{S}_2 = \Psi_{\mathscr{P}_w}(\mathscr{S}_1)$ that maps

$$\begin{array}{ll} turns_1(0) \mapsto (\bar{\mathbf{f}}, \bar{\mathbf{f}}) & turns_2(0) \mapsto (\bar{\mathbf{f}}, \bar{\mathbf{f}}) \\ turns_1(1) \mapsto (\overline{button_1(0)}, \bar{\mathbf{t}}) & turns_2(1) \mapsto (\overline{button_2(0)}, \bar{\mathbf{t}}) \end{array}$$

Another application refinement then results in the partial symbolic interpretation $\mathscr{S}_3 = \Psi_{\mathscr{P}_w}(\mathscr{S}_2)$ that maps

$$\begin{array}{ll} turns_1(0) \mapsto (\bar{\mathbf{f}}, \bar{\mathbf{f}}) & turns_2(0) \mapsto (\bar{\mathbf{f}}, \bar{\mathbf{f}}) \\ turns_1(1) \mapsto (\overline{button_1(0) \vee button_2(0)}, \bar{\mathbf{t}}) & turns_2(1) \mapsto (\overline{button_2(0) \vee button_1(0)}, \bar{\mathbf{t}}) \end{array}$$

Finally, one last unfoundedness refinement results in the symbolic interpretation $\mathscr{A}_w$ that maps

$$\begin{array}{ll} turns_1(0) \mapsto \bar{\mathbf{f}} & turns_2(0) \mapsto \bar{\mathbf{f}} \\ turns_1(1) \mapsto \overline{button_1(0) \vee button_2(0)} & turns_2(1) \mapsto \overline{button_1(0) \vee button_2(0)} \end{array}$$

In Figure A.1 in online Appendix A, a circuit representation of $Th(\mathscr{A}_w)$ is depicted. In this circuit, the different layers correspond to the evolution of the lower bound in different steps in the computation of the parametrised well-founded model of $\mathscr{P}_w$ (unfoundedness refinements are not visualised). In Figure A.2, the circuit for this examples with time ranging from 0 to 2 is depicted.

*Example 4.14* (*Example 4.12 continued*)
Well-founded inductions also work for positive logic programs. Let $\mathscr{S}_0$ denote the least precise partial interpretation. Since $\mathscr{P}_s$ is positive, it holds for every $i$ and $X$ that

$$\Psi^i_{\mathscr{P}_s}(\mathscr{S}_0)(smokes(X)) = (\mathscr{T}^i_{\mathscr{P}_s}(\bot)(smokes(X)), \bar{\mathbf{t}}).$$

Hence, repeated application refinements yield the partial symbolic interpretation $(\mathscr{A}_s, \top)$. One final unfoundedness refinement then results in the parametrised well-founded model of $\mathscr{P}_s$, namely $\mathscr{A}_s$.

***Discussion***

The condition in Theorem 4.11 naturally raises the question "what happens if the parametrised well-founded model is *not* exact?". First of all, our techniques also work in this setting. Indeed, Theorem 3.6 (1) guarantees that instantiating the the parametrised well-founded model of $\mathscr{P}$ with a $\Sigma_p$-interpretation $I$ results in the $\Psi^I_{\mathscr{P}}$-well-founded fixpoint of $T^I_{\mathscr{P}}$.

*Example 4.15*
Let $\mathscr{P}_{NT}$ be the following logic program

$$\{\ a \leftarrow \neg b. \quad b \leftarrow \neg a. \quad c \leftarrow \neg b \quad c \leftarrow e. \quad d \leftarrow a \wedge \neg c.\ \}$$

with parameter symbol $e$ and defined symbols $a, b, c$ and $d$. The parametrised well-founded model of $\mathscr{P}_{NT}$ is then $\mathscr{S}_{NT}$ such that

$$\mathscr{S}_{NT}(a) = (\overline{\mathbf{f}}, \overline{\mathbf{t}}) \qquad \mathscr{S}_{NT}(b) = (\overline{\mathbf{f}}, \overline{\mathbf{t}}) \qquad \mathscr{S}_{NT}(c) = (\overline{e}, \overline{\mathbf{t}}) \qquad \mathscr{S}_{NT}(d) = (\overline{\mathbf{f}}, \overline{\neg e})$$

However, in this text we mainly focus on programs with an exact parametrised well-founded model. Corollary 3.6 guarantees that this condition is satisfied for all logic programs in which the standard well-founded model is two-valued. This kind of programs is common in applications for deductive databases (Abiteboul and Vianu 1991) and for representing inductive definitions (Denecker and Vennekens 2014). Classes that satisfy this condition include monotone and (locally) stratified logic programs (Przymusinski 1988).

This restriction is typically not satisfied by ASP programs, where stable semantics is used. However, it deserves to be stressed that there is a strong relationship between ASP programs and logic programs under the parametrised well-founded semantics. Most ASP programs, e.g., those used in ASP competitions, are so-called generate-define-test (GDT) programs. They consist of three modules. A generate module opens the search space (i.e., it introduces parameter symbols); a define module contains inductive definitions for which well-founded and stable semantics coincide (as argued by Denecker and Vennekens (2014)) and a test module consist of constraints. Denecker *et al.* (2012) have argued that a GDT program is the *monotone conjunction* of its different modules. Hence, our technique can be used to compile the *define* part of a GDT program. The example below illustrates that only compiling this part results in an interpretation that captures the meaning of this definition more closely, by preserving more structural information.

*Example 4.16 (Example 4.15 continued)*
The first two rules of $\mathscr{P}_{NT}$ encode a choice rule for $a$ (or $b$). The define module of this program is the program

$$\mathscr{P}_{def} = \{\ b \leftarrow \neg a. \quad c \leftarrow \neg b \quad c \leftarrow e. \quad d \leftarrow a \wedge \neg c.\ \}$$

with parameter symbols $a$ and $e$, and defined symbols $b, c$ and $d$. The parametrised well-founded model of $\mathscr{P}_{def}$ is the symbolic interpretation $\mathscr{A}_{def}$ such that

$$\mathscr{A}_{def}(b) = \overline{\neg a} \qquad \mathscr{A}_{def}(c) = \overline{a \vee e} \qquad \mathscr{A}_{def}(d) = \overline{a \wedge \neg(a \vee e)} = \overline{\mathbf{f}}$$

As can be seen, the parametrised well-founded model now contains the information that $d$ is false, independent of the value of the parameter symbols (independent of the choice made in the choice rules in the original example).

## 5 Algorithms

Based on the theory developed in the previous section, we now discuss practical algorithms for exact and approximate knowledge compilation of logic programs.

### 5.1 Exact knowledge compilation

The definition of a well-founded induction provides us with a fixpoint procedure to compute the parametrised well-founded model. Our algorithms are parametrised by a language $\mathscr{L}$, referred to as the *target language*; this can be any representation of propositional formulas. We describe our algorithm, which we call COMPILE($\mathscr{L}$), as a (non-deterministic) finite-state-machine. A *state* $\mathfrak{S}$ consists of an assignment of two formulas $\mathfrak{S}_t(q)$ and $\mathfrak{S}_p(q)$ in $\mathscr{L}$ (over vocabulary $\Sigma_p$) to each atom $q \in \Sigma_d$. Hence, a state $\mathfrak{S}$ corresponds to the partial symbolic interpretation $\mathscr{S}_{\mathfrak{S}} = (\mathscr{A}_t, \mathscr{A}_p)$ such that for each $q \in \Sigma_d$, $\mathscr{A}_t(q) = \overline{\mathfrak{S}_t(q)}$ and $\mathscr{A}_p(q) = \overline{\mathfrak{S}_p(q)}$. The *transitions* in our finite-state-machine are exactly those tuples of states $(\mathfrak{S}, \mathfrak{S}')$ such that $\mathscr{S}_{\mathfrak{S}'}$ is a $\Psi_{\mathscr{P}}$-refinement of $\mathscr{S}_{\mathfrak{S}}$.

We further restrict these transitions to *maximally precise* transitions: *application refinements* that refine $\mathscr{S}$ to $\Psi_{\mathscr{P}}(\mathscr{S})$ and *unfoundedness refinements* as described in Proposition 2.1. Furthermore, we propose to make the resulting finite-state-machine *deterministic* by prioritising application refinements over unfoundedness refinements since they are cheaper, i.e., they only require one application of $\Psi_{\mathscr{P}}$.

The final output of COMPILE($\mathscr{L}$) is a theory $Th(\mathscr{A})$ in $\mathscr{L}$, where $\mathscr{A}$ is the parametrised well-founded model of $\mathscr{P}$. When $\mathscr{L}$ denotes Boolean circuits, each application of $\Psi_{\mathscr{P}}$ adds a layer of Boolean gates over the circuits in $\mathscr{S}_s$. When $\mathscr{L}$ denotes a language with a so-called APPLY function (Van den Broeck and Darwiche 2015) (e.g., SDDs), each application of $\Psi_{\mathscr{P}}$ calls APPLY to conjoin or disjoin circuits from $\mathscr{S}_s$.

Figure 2 contains an example circuit for the smokers problem (Example 4.12). The different layers in the circuit correspond to different steps in a well-founded induction (or the least fixpoint computation). Our algorithm follows the well-founded induction as described in Example 4.14, by prioritising application refinements over unfoundedness refinements. Similarly, our algorithm also follows the well-founded induction from Example 4.13. During the execution, circuits to represent the upper and lower bounds are gradually built (layer by layer).

*Theorem 5.1*

Let $\mathscr{L}_{BC}$ be the language of Boolean circuits. The following hold: *(i)* COMPILE($\mathscr{L}_{BC}$) has polynomial-time complexity and *(ii)* the size of the output circuit of COMPILE($\mathscr{L}_{BC}$) is polynomial in the size of $\mathscr{P}$.

In the terminology of Darwiche and Marquis (2002), this means that Boolean circuits are *at least as succinct* as logic programs under the parametrised well-founded semantics. With other languages, for example when $\mathscr{L}$ denotes OBDDs or SDDs, our algorithm can take exponential time, and its output can take exponential space in the size of $\mathscr{P}$. This is not surprising given the fact these languages support many (co-)NP hard inference tasks in polynomial time. Because they support equivalence checking (which is convenient to detect fixpoints early) and have a practically efficient Apply function (Van den Broeck and Darwiche 2015), OBDDs and SDDs are excellent languages for use in Compile.

### 5.2 *Approximate knowledge compilation*

The above section provides us with a way to perform various types of inference on logic programs: we can compile any logic program into a target formalism suitable for inference (e.g., SDD for equivalence checking or weighted model counting, CNF for satisfiability checking, etc.). However, when working with large programs this approach will be infeasible, simply because compilation is too expensive. In this case, we often want to perform approximate knowledge compilation (Selman and Kautz 1996). Well-founded inductions provide us with the means to do this.

*Proposition 5.2*
Suppose the parametrised well-founded model of $\mathscr{P}$ is $(\mathscr{A}, \mathscr{A})$. Let $(\mathscr{A}_{i,1}, \mathscr{A}_{i,2})$ be a well-founded induction of $\Psi_{\mathscr{P}}$. Then for every $i$, $Th(\mathscr{A}_{i,1}) \models Th(\mathscr{A}) \models Th(\mathscr{A}_{i,2})$.

One application of approximate knowledge compilation is in approximate inference by weighted model counting (WMC) (Chavira and Darwiche 2008) for probabilistic logic programs (Fierens *et al.* 2015). Let $\varphi$ be a formula (query) over $\Sigma$ and $w$ a weight function on $\Sigma$. Then it follows immediately from Proposition 5.2 that

$$\mathrm{WMC}(Th(\mathscr{A}_{i,1}) \wedge \varphi, w) \leqslant \mathrm{WMC}(\mathscr{P} \wedge \varphi, w) \leqslant \mathrm{WMC}(Th(\mathscr{A}_{i,2}) \wedge \varphi, w).$$

As Compile($\mathscr{L}$) follows a well-founded induction, it can be stopped at any time to obtain an upper and lower bound on the weighted model count (and therefore on the probability of the query). In fact, Proposition 5.2 can be used to perform *any* (anti)-monotonic inference task approximately.

## 6 Conclusion

In this paper, we presented a novel technique for knowledge compilation of general logic programs; our technique extends previously defined algorithms for positive logic programs. Our work is based on the constructive nature of the well-founded semantics: we showed that the algebraical concept of a well-founded induction translates into a family of anytime knowledge compilation algorithms. We used this to show that Boolean circuits are at least as succinct as logic programs (under the parametrised well-founded semantics). Our technique also extends to Kripke-Kleene semantics and to other knowledge representation formalisms. Extending the implementation by Vlasselaer *et al.* (2015) to general logic programs and testing it on a set of benchmarks are topics for future work.

## References

Abiteboul, S. and Vianu, V. 1991. Datalog extensions for database queries and updates. *J. Comput. Syst. Sci. 43,* 1, 62–124.

Antic, C., Eiter, T. and Fink, M. 2013. Hex semantics via approximation fixpoint theory. In *Proceedings of LPNMR*. 102–115.

Asuncion, V., Lin, F., Zhang, Y. and Zhou, Y. 2012. Ordered completion for first-order logic programs on finite structures. *Artif. Intell. 177–179*, 1–24.

Ben-Eliyahu, R. and Dechter, R. 1994. Propositional semantics for disjunctive logic programs. *Ann. Math. Artif. Intell. 12,* 1-2, 53–87.

Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers 35*, 677–691.

Cadoli, M. and Donini, F. M. 1997. A survey on knowledge compilation. *AI Commun. 10,* 3-4, 137–150.

Chavira, M. and Darwiche, A. 2005. Compiling bayesian networks with local structure. In *Proceedings of IJCAI*. 1306–1312.

Chavira, M. and Darwiche, A. 2008. On probabilistic inference by weighted model counting. *Artif. Intell. 172,* 6-7, 772–799.

Darwiche, A. 2011. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of IJCAI*. 819–826.

Darwiche, A. and Marquis, P. 2002. A knowledge compilation map. *J. Artif. Intell. Res. (JAIR) 17*, 229–264.

Denecker, M., Lierler, Y., Truszczyński, M. and Vennekens, J. 2012. A Tarskian informal semantics for answer set programming. In *ICLP (Technical Communications)*. 277–289.

Denecker, M., Marek, V. and Truszczyński, M. 2000. Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In *Logic-Based Artificial Intelligence, Springer*. Vol. 597. 127–144.

Denecker, M., Marek, V. and Truszczyński, M. 2004. Ultimate approximation and its application in nonmonotonic knowledge representation systems. *Information and Computation 192,* 1 (July), 84–121.

Denecker, M. and Vennekens, J. 2007. Well-founded semantics and the algebraic theory of non-monotone inductive definitions. In *LPNMR*. 84–96.

Denecker, M. and Vennekens, J. 2014. The well-founded semantics is the principle of inductive definition, revisited. In *Proceedings of KR*. 22–31.

Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D. S., Gutmann, B., Thon, I., Janssens, G. and De Raedt, L. 2015. Inference and learning in probabilistic logic programs using weighted boolean formulas. *TPLP 15,* 3, 358–401.

Fitting, M. 2002. Fixpoint semantics for logic programming — A survey. *Theoretical Computer Science 278,* 1-2, 25–51.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of ICLP/SLP*. 1070–1080.

Huang, J. and Darwiche, A. 2005. On compiling system models for faster and more scalable diagnosis. In *Proceedings of AAAI*. 300–306.

Janhunen, T. 2004. Representing normal programs with clauses. In *Proceedings of ECAI*. 358–362.

Janhunen, T. 2006. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics 16,* 1-2, 35–86.

Janhunen, T., Niemelä, I. and Sevalnev, M. 2009. Computing stable models via reductions to difference logic. In *LPNMR*, E. Erdem, F. Lin, and T. Schaub, Eds. LNCS, vol. 5753. Springer, 142–154.

Kleene, S. C. 1938. On notation for ordinal numbers. *The Journal of Symbolic Logic 3,* 4, 150–155.

Lifschitz, V. and Razborov, A. A. 2006. Why are there so many loop formulas? *ACM Trans. Comput. Log. 7,* 2, 261–268.

Lin, F. and Zhao, J. 2003. On tight logic programs and yet another translation from normal logic programs to propositional logic. In *Proceedings of IJCAI*. 853–858.

Lin, F. and Zhao, Y. 2004. ASSAT: Computing answer sets of a logic program by SAT solvers. *AIJ 157,* 1-2, 115–137.

Lowd, D. and Domingos, P. 2008. Learning arithmetic circuits. In *Proceedings of UAI*. 383–392.

Marek, V. and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*. Springer-Verlag, 375–398.

Palacios, H., Bonet, B., Darwiche, A. and Geffner, H. 2005. Pruning conformant plans by counting models on compiled d-dnnf representations. In *Proceedings of ICAPS*. 141–150.

Pelov, N., Denecker, M. and Bruynooghe, M. 2007. Well-founded and stable semantics of logic programs with aggregates. *TPLP 7,* 3, 301–353.

Przymusinski, T. C. 1988. On the declarative semantics of deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 193–216.

Selman, B. and Kautz, H. A. 1996. Knowledge compilation and theory approximation. *J. ACM 43,* 2, 193–224.

Strass, H. 2013. Approximating operators and semantics for abstract dialectical frameworks. *AIJ 205*, 39–70.

Suciu, D., Olteanu, D., Ré, C. and Koch, C. 2011. Probabilistic databases.

Van den Broeck, G. and Darwiche, A. 2015. On the role of canonicity in knowledge compilation. In *Proceedings of AAAI*.

van Emden, M. H. and Kowalski, R. A. 1976. The semantics of predicate logic as a programming language. *J. ACM 23,* 4, 733–742.

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

Vlasselaer, J., Van den Broeck, G., Kimmig, A., Meert, W. and De Raedt, L. 2015. Anytime inference in probabilistic logic programs with $T_{\mathscr{P}}$-compilation. In *Proceedings of IJCAI*. Available on `https://lirias.kuleuven.be/handle/123456789/494681`.