Meta-level Representations in the IDP Knowledge Base System: Towards Bootstrapping Inference Engine Development

Bart Bogaerts, Joachim Jansen, Broes De Cat, Gerda Janssens, Maurice Bruynooghe, and Marc Denecker

Department of Computer Science, KU Leuven

Abstract. Declarative systems aim at solving tasks by running inference engines on a *specification*, to free its users from having to specify *how* a task should be tackled. In order to provide such functionality, declarative systems themselves apply complex reasoning techniques, and, as a consequence, the development of such systems can be laborious work. In this paper, we demonstrate that the declarative approach can be applied to *develop* such systems, by tackling the tasks solved inside a declarative system declaratively. In order to do this, a meta-level representation of those specifications is often required. Furthermore, by using the language of the system for the meta-level representation, it opens the door to *bootstrapping*: an inference engine can be improved using the inference it performs itself.
One such declarative system is the IDP knowledge base system, based on the language $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, a rich extension of first-order logic. In this paper, we discuss how $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ can support meta-level representations in general and which language constructs make those representations even more natural. Afterwards, we show how meta-$\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ can be applied to bootstrap its model expansion inference engine. We discuss the advantages of this approach: the resulting program is easier to understand, easier to maintain and more flexible.

1 Introduction

Declarative systems aim at solving tasks by running inference engines on a *specification*, to free its users from having to prescribe *how* a task should be tackled. Many computational tasks can be represented more easily in a declarative language than in an algorithmic fashion. Declarative languages and systems are developed, e.g., in the fields of Functional Programming (FP) [5], Constraint Programming (CP) [3] and Logic Programming (LP) [22, 23], including Answer Set Programming (ASP) [4, 16]. To be able to apply inference on declarative specifications, these systems need to apply complex reasoning techniques, and, as a consequence, their development can be laborious work. This can be observed for example from the decreasing number of ASP systems participating in ASP competitions [2, 7, 15] and in the relatively large number of available ASP solvers

compared to few available ASP grounders, as the latter work on a much richer input language.

In this paper, we demonstrate that parts of such systems can themselves be implemented declaratively, offering the same advantages to the developer as to the user of the declarative system, such as reduced development time, increased flexibility and easier maintenance. In order to do this, a meta-level representation of those specifications is often required. Meta-programming is well-known from both declarative languages, such as the built-in meta-predicates in Prolog [1], and procedural languages, such as the template meta-programming component in C++ [25]. Recently, meta-programming has been applied in ASP. For example, Gebser et al. [18] present a meta-approach to declaratively debug ASP programs. In [19], a meta-level ASP representation is used to manage interacting embedded computational objects that publish their properties as ASP programs.

We develop our ideas in the context of IDP [6, 8], a knowledge-base system (KBS) [13] for the logic FO$(\cdot)^{\texttt{IDP}}$, a language that extends first-order logic (FO) with, among others, inductive definitions and a type system. The system supports a range of inference tasks, such as querying, model expansion, optimisation, propagation and deduction. In this paper we will show how to bootstrap IDP: how to improve the inference performed by IDP, using the inference performed by IDP itself.

First, we discuss how to support meta-level representations in FO$(\cdot)^{\texttt{IDP}}$. We start by describing a highly general representation for propositional logic. In this representation, all formulas are represented using constructor functions, such as a function *and* that maps two formulas to their conjunction. For example, the formula $p \vee (q \wedge \neg r)$ is represented by the term

$$or(atom(p), and(atom(q), not(atom(r)))).$$

Afterwards, we discuss how to extend this approach to full FO$(\cdot)^{\texttt{IDP}}$. In this discussion, we research language features that would make the representation even more natural, such as sets. Furthermore, we argue that, even without those extensions, the current finite domain solvers simply cannot do meta-level reasoning because the domains are always infinite. Indeed, a total function such as *and* that maps two formulas to their conjunction immediately results in an infinite domain. Therefore, we discuss several solutions to solve the problems with infinite domains. One of these solutions consists of representing a logical specification by its parse tree: now the domain consists of a finite set of nodes of parse trees and we use a new meta-vocabulary, where objects are nodes in the parse tree of a specific theory. The above formula $(p \vee (q \wedge \neg r))$ is now represented by a set of parse-tree nodes. Each of these nodes is augmented with additional information, such as their type (conjunctive node, disjunctive node, negation, atom, ...) and with links to their subformulas. More concretely, we represent

this formula by the following first-order structure:

$Symbol = \{p, q, r\} \quad Formula = \{\varphi_1, \ldots, \varphi_6\}$
$Subform = \{(\varphi_1, 1) \mapsto \varphi_4, (\varphi_1, 2) \mapsto \varphi_2, (\varphi_2, 1) \mapsto \varphi_5, (\varphi_2, 2) \mapsto \varphi_3, (\varphi_3, 1) \mapsto \varphi_6\}$
$Kind_F = \{\varphi_1 \mapsto disj, \varphi_2 \mapsto conj, \varphi_3 \mapsto neg, \varphi_4 \mapsto atom, \varphi_5 \mapsto atom, \varphi_6 \mapsto atom\}$
$Symbol_F = \{\varphi_3 \mapsto p, \varphi_4 \mapsto q, \varphi_5 \mapsto r\}$

The above structure represents a disjunctive formula (φ_1) with two subformulas (φ_4 and φ_2 respectively). The first of these subformula is an atom, namely p, etcetera. From a representation point of view, we prefer the first solution, where we use the infinite domains as they result in clear and simple representations. However, from a practical point of view, we prefer the second solution, where only relevant nodes in the parse tree are taken into account. Conceptually, the latter solution is similar to the approach taken in many applications of meta-reasoning in the context of ASP [17–19]. The biggest difference is that in ASP, one typically enforces restrictions on programs to handle infinite domains: programs are assumed to be safe, i.e., written by a careful programmer to ensure that the grounding is finite, while we allow any FO$(\cdot)^{\text{IDP}}$ theory.

Afterwards, we show how a subtask of model expansion, can be modelled on the meta-level and solved using the model expansion engine itself. This allows us to effectively *bootstrap* parts of our engine. This reduces development effort and yields more flexible, bug-free, and maintainable code. The application we discuss consists of finding an optimal strategy to split a theory in many pieces that can be handled individually, or pieces that can safely be ignored by the model expansion engine. Model expansion is closely related to constraint programming and answer set generation, as discussed in [11]. Hence, this application, and similar meta-level bootstrapping applications, can also be applied in the context of CP and ASP systems. Solving them declaratively could also reduce development effort in those fields.

The main contributions of this paper are that it introduces and compares several solutions to meta-modelling in FO$(\cdot)^{\text{IDP}}$ and presents a new application of meta-modelling, and shows how this application can be used to bootstrap declarative systems.

The rest of the paper is structured as follows. In Section 2, we review relevant concepts and notations. In Section 3, we present meta-level FO$(\cdot)^{\text{IDP}}$ representations and discuss their properties. In Section 4, we present a detailed application of the meta-level representations. In Section 5, we briefly discuss several other bootstrapping applications that can be used to optimise IDP. In Section 6, we discuss the implementation of the application. Concluding remarks and future work follows in Section 7.

2 Preliminaries

In this section we present the language FO$(\cdot)^{\text{IDP}}$, focusing on the aspects that are relevant for this paper. Details and examples can be found in [6, 8]. We assume familiarity with basic concepts of FO. FO$(\cdot)^{\text{IDP}}$ is a many-typed logic;

thus, a *vocabulary* Σ is a set of type, predicate, and function symbols. We write $p[t_1, \ldots, t_n]$ and $f[t_1, \ldots, t_n \to t']$ for the predicate p with arguments of type $t_1, \ldots, t_n$, respectively the function f with input arguments of type $t_1, \ldots, t_n$ and an output argument typed t'. We use $\forall x \in t : \varphi$ and $\exists x \in t : \varphi$ to indicate that x is quantified over the elements in type t. When introducing a predicate $p[t_1, \ldots, t_n]$, we often immediately define its informal semantics. In order to do this, we write $p[x_1 : t_1, \ldots, x_n : t_n]$ with a natural language sentence explaining the meaning of an atom $p(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are variables of the correct type. For example "we use a predicate $r[x : t, y : t]$ to express that y is reachable from x in the graph at hand". A *domain atom* is an expression of the form $P(\overline{d})$ or of the form $f(\overline{d}) = d'$, where P is a predicate symbol, f is a function symbol and the d are domain elements. Structures can be *three-valued*. Concretely, this means that a structure assigns a truth value true (**t**), false (**f**) or unknown (**u**) to every domain atom. If I is a structure over a vocabulary Σ, and σ a symbol in Σ, σ^I denotes the interpretation of σ in I. In FO$(\cdot)^{\text{IDP}}$, one can declare a type t as a *constructed type*, a type constructed from a finite set of constructor function symbols $\{f[t_1, \ldots, t_n \to t], \ldots, g[s_1, \ldots, s_m \to t]\}$. Concretely, these constructors are total injective functions, have disjoint ranges and the union of their ranges is (the interpretation of) t. For example, we can define the infinite type *List* representing lists of integers as the type constructed from $nil[\to List]$ and $cons[int, List \to List]$. In the untyped case, this corresponds to the condition that the domain is the Herbrand universe.

FO$(\cdot)^{\text{IDP}}$ extends FO with (inductive) definitions: sets of rules of the form $\forall \overline{x} : p(\overline{t}) \leftarrow \varphi$, (or $\forall \overline{x} : f(\overline{t}) = t' \leftarrow \varphi$) where φ is an FO formula and the free variables of φ and $p(\overline{t})$ are among the $\overline{x}$. We call $p(\overline{t})$ (respectively $f(\overline{t}) = t'$) the head of the rule and φ the body. The connective $\leftarrow$ is the *definitional implication*, which should not be confused with the material implication $\Rightarrow$. Thus, the expression $\forall \overline{x} : p(\overline{t}) \leftarrow \varphi$ is *not* a shorthand for $\forall \overline{x} : p(\overline{t}) \vee \neg\varphi$. Instead, its meaning is given by the well-founded semantics (for functions, semantics of the graph predicate is considered, i.e., as if the rule were $graph_f(\overline{t}, t) \leftarrow \varphi$); this semantics, for example, correctly formalises all kinds of definitions that typically occur in mathematical texts [12, 14]. An FO$(\cdot)^{\text{IDP}}$ theory consists of a set of FO sentences and definitions.

The *model expansion* task takes as input a theory $\mathcal{T}$ and structure I, both over a vocabulary Σ, and vocabulary σ_{out}, subset of Σ. We denote such a task as $MX\,\langle \Sigma, \mathcal{T}, I, \sigma_{out} \rangle$. The aim is to find two-valued σ_{out}-structures for which a Σ-expansion exists that is more precise than I and is a model of $\mathcal{T}$. This task corresponds to finding, e.g., a partial solution to a constraint satisfaction problem with the guarantee that a total solution exists.

3 Approaches for Meta-Reasoning: Problems and Solutions

In order to develop a meta-language to reason about FO$(\cdot)^{\text{IDP}}$, there are several possibilities, each with their own advantages and disadvantages. In Section 3.1,

we first describe a highly general, flexible approach to meta-modelling propositional logic. In Section 3.2, we discuss how to extend this method to FO$(\cdot)^{\text{IDP}}$. Afterwards, in Sections 3.3 and 3.4, we discuss alternative approaches that are less flexible, but more efficient to work with. The application presented in Section 4 uses those more efficient approaches.

3.1 A General Approach to Meta-Modelling Propositional Calculus

The approach we discuss in this section, which we call the *Herbrand approach*, is based on constructor functions (see Section 2). Consider for example an alphabet Σ of propositional symbols. The language $\mathcal{L}$ of propositional formulas over Σ is then defined as follows

- if $p \in \Sigma$, then p is a formula,
- if φ and ψ are formulas, then $\varphi \wedge \psi$ is a formula.
- if φ and ψ are formulas, then $\varphi \vee \psi$ is a formula,
- if φ is a formula, then $\neg\varphi$ is a formula.

To represent propositional formulas with constructors in a typed language, we use two types: *Symbol* and *Formula* (abbreviated below as S and F, respectively). The type *Formula* is constructed from four constructor functions, one for each of the above rules, namely $atom[S \to F]$, $and[F, F \to F]$, $or[F, F \to F]$ and $not[F \to F]$. Hence, every structure interprets F as the union of the images of each of these four functions.

Example 1. The formula $p \vee (q \wedge \neg r)$ is represented by the term

$$or(atom(p), and(atom(q), not(atom(r))))$$

with *Symbol* interpreted as (a superset of) $\{p, q, r\}$.

This approach has many advantages. First of all, it is very generic and flexible. All interesting information about a formula can be derived from a representation as above. For example, if we want to define a relation $subfOf[F, F]$ that expresses that a formula is a (direct or indirect) subformula of another, we do this as follows:

$$\left\{\begin{array}{ll} \forall x: & subfOf(x, x). \\ \forall x, y: & subfOf(x, and(x, y)). \quad \forall x, y: subfOf(y, and(x, y)). \\ \forall x, y: & subfOf(x, or(x, y)). \quad \forall x, y: subfOf(y, or(x, y)). \\ \forall x: & subfOf(x, not(x)). \\ \forall x, y: & subfOf(x, y) \leftarrow \exists z: subfOf(x, z) \wedge subfOf(z, y). \end{array}\right\}$$

Second, since Herbrand interpretations are used, all formulas over the available symbols are in the domain. This allows reasoning about formulas not explicitly in the input. For example, the transformation $nnf[F \to F]$ that maps formulas to

their equivalent formula in Negation Normal Form (NNF) by pushing negations inwards can be defined as follows

$$\left\{ \begin{array}{l} \forall x : nnf(not(not(x))) = nnf(x). \\ \forall x, y : nnf(not(and(x, y)) = or(nnf(not(x)), nnf(not(y)). \\ \qquad\qquad \vdots \end{array} \right\}$$

In order to use this meta-modelling approach for bootstrapping (a propositional version of) IDP, one needs to implement two transformations: converting internal data structures that represent a logical specification into a structure in the meta-vocabulary and back again. Given these two transformations, one can use inference on the structure that represents a specification, obtain a new structure (e.g., after applying model expansion) and successively transform it back into internal data structures.

3.2 A General Approach to Meta-Modelling FO(·)$^{\texttt{IDP}}$

The above approach using Herbrand interpretations can be extended to represent full first-order formulas, or, more general, FO(·)$^{\texttt{IDP}}$ theories. It is, however, less trivial to do this in a nice and principled way. Consider, e.g., how to represent:

- An FO(·)$^{\texttt{IDP}}$ theory, which is a *set* of FO sentences and inductive definitions,
- A first-order atom, which is a predicate symbol applied to a *list* of terms.

For this, we need to be able to represent sets or lists (of unknown size) in some way. We see three solutions. Either, we use type theory or higher-order logic to integrate the notions of sets and/or lists in the language. Or, we use infinitely many constructors, i.e., one for each possible arity. Or, as a third possibility, we encode lists in some way, for example as Prolog-like head-tail lists. However, all three solutions have some disadvantages. We discuss these in the following two sections and show how they can be addressed to obtain a feasible meta-modelling approach.

3.3 Obtaining Finite Domains

From a knowledge representation point of view, the approach using constructor functions is the best solution. However, this approach has some practical disadvantages. The most important one is that, since all formulas and terms over Σ are part of the domain, the domain immediately is infinite, thus no finite domain solvers can be used to perform inference on such specifications. Searching for models of a theory over an infinite domain often requires smart forms of reasoning. It is ongoing research to handle infinite domains better [10], but this is far from finished. Hence, in this paper we provide alternatives to the infinite encodings.

In order to obtain finite domains, we consider several solutions. One solution (**i**) is to restrict our attention to formulas and terms occurring in the input theory. This can be achieved by simply restricting types such as *Formula* to relevant

objects. This implies that all of the constructor functions now become partial functions, e.g., $and(\varphi, \psi)$ is only defined if the conjunction of φ and ψ occurs in the input theory. Representationally, this approach is still very similar to the Herbrand approach. Furthermore, it works well for applications that are concerned with analysis, e.g., checking whether the input theory satisfies a certain criterion. However, sometimes we need to reason about formulas not in the input theory; in this case, things get more difficult. For example, the above-mentioned application to push negations inward requires more formulas than only the ones that occur in the original theory. Other solutions to avoid infinite domains are then (**ii**) to overestimate the number of additional required domain elements or (**iii**) to iteratively perform the inference with more domain elements.

To apply (**ii**) or (**iii**), we need to be able to use domain elements with unknown properties. For example, in order to transform a formula to NNF, we are searching for a formula that is equivalent with the original one, but that is in NNF. Of course, we do not know in advance which formula this is. Now, we note that when restricting the Herbrand approach to formulas occurring in the input, the domain elements are basically nodes in the parse tree of a specification. For each of these nodes, certain information is known. For example the node $and(\varphi, \psi)$ is a node representing a conjunctive formula, with subformulas φ and ψ. Our alternative representation approach is based on that observation: domain elements in *Formula* are nodes of a parse trees and certain information about those nodes is known.

To define our alternative meta-approach, we start from scratch. An $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ vocabulary Σ is represented with a meta-vocabulary consisting of the following types:

- *Type*, containing all types in Σ,
- *Symbol*, containing all predicate and function symbols in Σ,
- and *Index*, to refer to the possible argument positions.

Furthermore, we use a function $arity[s : Symbol \rightarrow n : Index]$, which expresses that s has arity n, and a partial function $type[s : Symbol, i : Index \rightarrow t : Type]$, such that t is the i'th type in the type signature of s. For n-ary function symbols, the function *type* maps index $n + 1$ to the output type of f.

In order to also represent $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ theories, we add the following types:

- *Formula*, to represent formulas in the domain of interest,
- *Variable*, to represent variables,
- and *Term*, to represent terms in the domain of interest.

For each formula and term in the domain, their properties must also be described as part of the meta-specification. For example, a formula $\forall x : P(x)$ is a universal quantification over variable x with subformula $P(x)$. In order to represent these properties we use:

- a type $Kind_{Formula}$ that is used to distinguish the different kinds of formulas; this type has a fixed interpretation consisting of $conj$, $disj$, $quant_{univ}$, $atom$, etc.,

- a function $kind_{Formula}[f : Formula \rightarrow t : Kind_{Formula}]$ that maps every formula f to the kind of formula it represents (for example, the formula $\forall x : P(x)$ would be mapped to $quant_{univ}$),
- a type $Kind_{Term}$, similar to $Kind_{Formula}$, that contains all kinds of terms. In our case, these are $domElem$ (domain elements), var (variables) or $functerm$ (terms obtained by function application),
- a function $kind_{Term}[t : Term \rightarrow k : Kind_{Term}]$ such that k is the kind of term represented by t,
- partial functions that map formulas to their constituent components, such as $symbol_{Formula}[f : Formula \rightarrow s : Symbol]$, that maps atoms f to their predicate symbol s, and $subform[f : Formula, i : Index \rightarrow sf : Formula]$, which indicates that sf is the i'th subformula of f (for example a quantification has one subformula and a conjunction has two). The meta-vocabulary contains similar functions for terms and subterms.

In order to represent rules, we add a type *Rule* with two functions $head[Rule \rightarrow Formula]$ and $body[Rule \rightarrow Formula]$ mapping a rule to their head and body respectively. Definitions (sets of rules) and theories (sets of definitions and sentences) are encoded with predicates $in[e{:}\,Rule, s{:}\,Definition]$, $in[e{:}\,Definition, s{:}\,Theory]$ and $in[e{:}\,Formula, s{:}\,Theory]$ meaning that the object e is an element of the set s. Similar types and functions can straightforwardly be defined to also represent structures, but this is not necessary in this paper. Below, we often abbreviate *Formula* to F and *Term* to T.

One advantage of this approach is that we can add extra domain elements to F and T, for example add an extra formula, without fixing their properties in advance. A reasoning engine can then assign kinds and properties to these extra symbols. For example, for the NNF transformation from the previous section, if the input is $\neg(P \vee Q)$, the input theory consists of four formulas (P, Q, $P \vee Q$ and $\neg(P \vee Q)$). In order to transform it to NNF, we add 4 extra domain elements to F and leave their properties open (one to represent the NNF form of each of the formulas). The system only needs to use three of those four extra placeholders, namely to represent the formulas $\neg P$, $\neg Q$ and $\neg P \wedge \neg Q$.

Example 2. If $p[T, T]$ is a predicate and $f[T \rightarrow T]$ a function, then a formula $\forall x[T] : p(x, f(x))$ is represented as follows. The information about the symbols is represented by

$$\begin{aligned} Type &= \{T\}, Symbol = \{p, f\}, \\ Index &= \{1, 2\}, \\ arity &= \{p \mapsto 2, f \mapsto 1\}, \\ type &= \{(p,1) \mapsto T,\ (p,2) \mapsto T,\ (f,1) \mapsto T,\ (f,2) \mapsto T\} \end{aligned}$$

In addition, we introduce domain elements φ_1 and φ_2 to represent the formulas $\forall x \in T : p(x, f(x))$ and $p(x, f(x))$, and domain elements t_1, t_2 and t_3 to represent the term occurrences x (first occurrence), respectively $f(x)$ and the second

occurrence of x. The remaining information can then be encoded as follows.

$$
\begin{aligned}
&\mathit{Formula} = \{\varphi_1, \varphi_2\},\ \mathit{Term} = \{t_1, t_2, t_3\}, \mathit{Variable} = \{x\},\\
&\mathit{kind}_F \ = \{\varphi_1 \mapsto \mathit{quant}_{univ},\ \varphi_2 \mapsto \mathit{atom}\},\\
&\mathit{kind}_T \ = \{t_1 \mapsto \mathit{var},\ t_2 \mapsto \mathit{functerm},\ t_3 \mapsto \mathit{var}\},\\
&\mathit{subform} = \{(\varphi_1, 1) \mapsto \varphi_2,\ (\varphi_2, 1) \mapsto t_1,\ (\varphi_2, 2) \mapsto t_2\},\\
&\mathit{symbol}_F = \{\varphi_2 \mapsto p\}, \mathit{symbol}_T = \{t_2 \mapsto f\},\\
&\ldots
\end{aligned}
$$

In order to implement bootstrapping applications using the above approach we need to translate internal data structures into structures over the above vocabulary (and back) and come up with methods to approximate the number of extra required domain elements of every sort.

3.4 Abstractions

The previous sections describe a very detailed way to represent FO$(\cdot)^{\texttt{IDP}}$ theories. For most applications, such a detailed representation is not necessary and an abstraction of the detailed information suffices. For example, in Section 4, we describe an application where an optimal model expansion workflow is computed for a given theory. The only input it requires is

- which symbols are defined in which definitions,
- which symbols are open in which definitions, and
- which symbols occur in first-order sentences.

In the presentation of all of our applications, we assume a representation at the right level of abstraction is available a priori. In Section 4.1, we will show that obtaining input in the right abstraction level can itself be cast as a bootstrapping task.

4 Splitting the Model Expansion Task

As explained above, the model expansion task takes as input a theory $\mathcal{T}$ and structure I, both over a vocabulary Σ, and vocabulary σ_{out}, subvocabulary of Σ. The aim is to find two-valued σ_{out}-structures for which a Σ-expansion exists that is more precise than I and is a model of $\mathcal{T}$. By default, the IDP system uses the ground-and-solve technique to solve model expansion problems. However, for many special cases more efficient techniques exist. One important challenge is to detect these special cases as often as possible. For example, calculating the well-founded model of a definition whose parameters are known corresponds to querying a logic program, a task that has been studied intensively, and has been implemented in various Prolog systems. Jansen et al. [20] show that great performance gains are possible by splitting the model expansion task: first evaluate all definitions whose open predicates are interpreted using dedicated techniques and subsequently use ground-and-solve for the rest of the theory. Or, as a second example, given the output vocabulary, one can often ignore parts of the theory that are not relevant for the problem at hand.

Example 3. As a small example, consider a theory consisting of the definitions $\Delta_1 = \{p \leftarrow q\}$, $\Delta_2 = \{q \leftarrow r\}$ and $\Delta_3 = \{t \leftarrow r \wedge s\}$ and the single FO sentence $q \vee s$. If model expansion on this theory is performed with an input structure that interprets r, Δ_2 can be evaluated beforehand, hence the value of q can be determined before grounding and solving. This implies in turn that also Δ_1 can be evaluated in advance. We can go further than this: evaluation of Δ_1 and of Δ_3 can be postponed until after the search since symbols p and t do not occur in the FO sentence and are irrelevant for the search. Furthermore, if t is not part of the output vocabulary σ_{out}, definition Δ_3 does not need to be evaluated at all.

Summarised, for a model expansion task with theory $\mathcal{T}$, structure $\mathcal{I}$ and output-vocabulary σ_{out}, we partition the set of definitions in $\mathcal{T}$ into four parts:

- **Preprocess**: Definitions that can be evaluated before grounding and solving the theory.
- **Postprocess**: Definitions that can be evaluated after search.
- **Forget**: Definitions that are irrelevant for this model expansion problem.
- **Search**: Definitions without special properties, i.e., that should be considered during search.

The definition we preprocess are highly similar to domain predicates in the ASP grounder lparse [24]. In lparse, some parts of an ASP program (some definitions, as we would say) are also evaluated prior to grounding. We extended these ideas to also postprocess and/or forget some parts of the theory.

We modelled the partition of the definitions in an FO($\cdot$) theory in the above four classes of definitions using the approach described in Section 3.4. The input for this partition problem is a structure interpreting the following symbols (hence, an abstraction of the most precise meta-representation):

- types: *Def* and *Symbol*,
- relations $open[d : Def, s : Symbol]$ and $def[d : Def, s : Symbol]$, with intended interpretation that s is open (respectively defined) in d,
- a relation $occursInFO[s : Symbol]$, meaning that s occurs in FO constraints,
- a relation $twoVal[s : Symbol]$ meaning that s is two-valued in the input structure of the model expansion task,
- a relation $output[s : Symbol]$ meaning that s is an element of the output-vocabulary σ_{out}.

The output of the problem then consists of the relations $pre[Def]$, $search[Def]$, $post[Def]$ and $forget[Def]$, describing the definitions to preprocess, use for search, postprocess, and forget respectively. Furthermore, in the theory below, we use auxiliary relations

- $hasOpen[s_1 : Symbol, s_2 : Symbol]$ meaning that some definition of s_1 has s_2 as open,
- $outRel[s : Symbol]$ meaning that the value of s is relevant for computing the values of all symbols in the output vocabulary, and

– *searchIrrel*[s : *Symbol*] meaning that the value of s is irrelevant for the search problem.

The following theory then describes an optimal splitting for step 3 of the model expansion workflow, where definitions are either ignored or evaluated in post- and preprocessing steps as much as possible:

$$
\left\{
\begin{array}{ll}
\forall s, s' : hasOpen(s, s') & \leftarrow \exists d : def(d, s) \wedge open(d, s'). \\
\forall s : outRel(s) & \leftarrow output(s). \\
\forall s : outRel(s) & \leftarrow \exists s' : hasOpen(s', s) \wedge outRel(s'). \\
\forall s : searchIrrel(s) & \leftarrow \neg occursInFO(s) \\
 & \quad \wedge \#\{d \mid def(d, s)\} \leq 1 \\
 & \quad \wedge \forall s' : hasOpen(s', s) \Rightarrow searchIrrel(s').
\end{array}
\right\}
$$

$$
\left\{
\begin{array}{ll}
\forall d : pre(d) & \leftarrow (\exists s : \neg searchIrrel(s) \wedge def(d, s)) \\
 & \quad \wedge (\forall s : open(d, s) \Rightarrow \\
 & \quad\quad (twoVal(s) \vee \exists d' : def(d', s) \wedge pre(d'))). \\
\forall d : search(d) & \leftarrow \neg pre(d) \wedge \exists s : \neg searchIrrel(s) \wedge def(d, s). \\
\forall d : post(d) & \leftarrow (\exists s : def(d, s) \wedge outRel(s)) \wedge \neg pre(d) \wedge \neg search(d). \\
\forall d : forget(d) & \leftarrow \neg pre(d) \wedge \neg search(d) \wedge \neg post(d).
\end{array}
\right\}
$$

This theory states that there are two conditions on definitions in order to preprocess them: one of their defined symbols is relevant for the search part, and all of its opens are either input or calculable from input. We ground-and-solve all definitions that define constrained symbols (symbols that are relevant for the search) unless they are already preprocessed. Furthermore, we postprocess all definitions that define a relevant symbol (one that is either in the output vocabulary or is needed to evaluate all symbols from the output vocabulary) and forget all other definitions. In [21], it has been shown that removing redundant information from a theory can influence the efficiency of the underlying solver. However, the information we remove does not have similar side effects, since we only remove (or postpone) definitions of symbols that are irrelevant for the search part.

The application we presented is an analysis application: we analyse the structure of a set of definitions given in the input structure. Hence, this application does not need the introduction or invention of new values.

4.1 Marshalling

In Section 3.4 we mentioned that for many applications, an abstraction of the specification suffices. Such abstraction can themselves be obtained through bootstrapping. For example, in the above bootstrapping application, we silently assumed an input that describes

– which symbols are defined in which definitions,

– which symbols are open in which definitions, and
– which symbols occur in first-order sentences.

Obtaining such abstraction, as described in Section 3.4 starting from the most detailed representation can be done using definitions. For example, in this application, defining which symbols occur in first-order sentences in a theory identified with constant T is done with

$$\left\{\begin{array}{l} \forall s : \mathit{occursInFO}(s) \leftarrow \exists f[F] : \mathit{in}(f, T) \wedge \mathit{occursIn}(s, f). \\ \forall s, f : \mathit{occursIn}(s, f) \leftarrow \mathit{kind}_F(f) = \mathit{atom} \wedge \mathit{symbol}(f) = s. \\ \forall s, f : \mathit{occursIn}(s, f) \leftarrow \exists f' : \mathit{subfOf}(f', f) \wedge \mathit{occursIn}(s, f'). \end{array}\right\}$$

Given a definition that describes an abstraction, definition evaluation yields the desired abstraction, hence the marshalling is done using bootstrapping as well.

5 More Bootstrapping Applications

The bootstrapping application discussed above is only the tip of the iceberg. There are, for example, many different transformations that could be handled in a declarative way. Also several theoretical results, e.g., results about splitting logic programs or inductive definitions [12], can easily be reformulated in the meta-vocabulary. Another task currently solved with bootstrapping and meta-modelling in IDP is in the context of lazy grounding: given a theory, find a maximally large part of the theory of which the grounding can be delayed. It would take us too far to present this application here, details can be found in Section 4.2 of De Cat et al.'s paper on lazy grounding [10].

Furthermore, in this paper we focused on an application that can be cast as a model expansion task. In a KBS, there are many more bootstrapping opportunities. For example the query inference is used to compute the value of optimisation terms during optimal model expansion and the deduction inference is used to detect implicit functional dependencies [9].

6 Discussion

In the previous sections, we presented several ideas concerning meta-level representations of FO$(\cdot)^{\texttt{IDP}}$ and bootstrapping applications that use these meta-level representations. We implemented some of these ideas in the IDP system. More concretely, the application described in Section 4 is now by default part of IDP's model expansion workflow, implemented using bootstrapping. Of course, for all bootstrapping applications, one can argue that dedicated algorithms to perform these tasks can be more efficient than the bootstrapping approach which applies a generic model expansion engine. However, the complexity typically scales with the size of the input structure, which is relatively small as it is actually the encoding of a theory. Implementing the other ideas presented in Section 5 is part of future work.

When bootstrapping model expansion by solving subtasks with model expansion themselves, we have to be cautious to avoid infinite loops. These loops are avoided by making the subtasks optional: the user (or the programmer in case of bootstrapping) can, for example, inform the system that the theory does not need to be partitioned in the four components described in Section 4.

In our current implementation, we still write out information directly at the abstraction level of the application. It is part of future work to start from the most general meta-representation and apply the marshalling techniques described in Section 4.1 instead.

7 Conclusion

Declarative systems solve complex reasoning tasks over a general input. In addition, they aim at not burdening the user with performance considerations during modelling. As a result, implementing them is laborious, as they need to handle many special cases to guarantee reasonable efficiency. In addition, it is difficult to reuse such optimisations over different systems without a lot of implementation work.

In this paper, we showed that tasks solved within a declarative system can also be solved declaratively. We presented meta-modelling approaches for $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ to tackle such tasks and an application that is used to bootstrap model expansion.

These techniques are actively used within IDP to reduce development time and obtain more flexible, bug-free, and maintainable code. Furthermore, these bootstrapping techniques cause improvements to one engine to have a positive effect on the whole system: small improvements to, e.g., its model expansion engine can result in improved efficiency for all tasks where model expansion is applied internally.

References

1. Abramson, H., Rogers, H.: Meta-programming in logic programming. MIT Press (1989)
2. Alviano, M., Calimeri, F., Charwat, G., Dao-Tran, M., Dodaro, C., Ianni, G., Krennwallner, T., Kronegger, M., Oetsch, J., Pfandler, A., Pührer, J., Redl, C., Ricca, F., Schneider, P., Schwengerer, M., Spendier, L.K., Wallner, J.P., Xiao, G.: The fourth Answer Set Programming competition: Preliminary report. In Cabalar, P., Son, T.C., eds.: LPNMR. Volume 8148 of LNCS., Springer (2013) 42–53
3. Apt, K.R.: Principles of Constraint Programming. Cambridge University Press (2003)
4. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, New York, NY, USA (2003)
5. Bird, R.: Pearls of Functional Algorithm Design. Cambridge University Press (2010)

6. Bruynooghe, M., Blockeel, H., Bogaerts, B., De Cat, B., De Pooter, S., Jansen, J., Labarre, A., Ramon, J., Denecker, M., Verwer, S.: Predicate logic as a modeling language: Modeling and solving some machine learning and data mining problems with IDP3. TPLP ((in press) 2014)
7. Calimeri, F., Ianni, G., Ricca, F.: The third open Answer Set Programming competition. CoRR **abs/1206.3111** (2012)
8. De Cat, B., Bogaerts, B., Bruynooghe, M., Denecker, M.: Predicate logic as a modelling language: The IDP system. CoRR **abs/1401.6312** (2014)
9. De Cat, B., Bruynooghe, M.: Detection and exploitation of functional dependencies for model generation. TPLP **13**(4-5) (2013) 471–485
10. De Cat, B., Denecker, M., Stuckey, P.J., Bruynooghe, M.: Lazy model expansion: Interleaving grounding with search. CoRR **abs/1402.6889** (2014)
11. Denecker, M., Lierler, Y., Truszczynski, M., Vennekens, J.: A Tarskian informal semantics for answer set programming. In Dovier, A., Costa, V.S., eds.: ICLP (Technical Communications). Volume 17 of LIPIcs., Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012) 277–289
12. Denecker, M., Ternovska, E.: A logic of nonmonotone inductive definitions. ACM Trans. Comput. Log. **9**(2) (April 2008) 14:1–14:52
13. Denecker, M., Vennekens, J.: Building a knowledge base system for an integration of logic programming and classical logic. In García de la Banda, M., Pontelli, E., eds.: ICLP. Volume 5366 of LNCS., Springer (2008) 71–76
14. Denecker, M., Vennekens, J.: The well-founded semantics is the principle of inductive definition, revisited. In: KR. ((in press) 2014)
15. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second Answer Set Programming competition. In Erdem, E., Lin, F., Schaub, T., eds.: LPNMR. Volume 5753 of LNCS., Springer (2009) 637–654
16. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers (2012)
17. Gebser, M., Kaminski, R., Schaub, T.: Complex optimization in Answer Set Programming. TPLP **11**(4-5) (2011) 821–839
18. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A meta-programming technique for debugging answer-set programs. In Fox, D., Gomes, C.P., eds.: AAAI, AAAI Press (2008) 448–453
19. Janhunen, T., Luukkala, V.: Meta programming with answer sets for smart spaces. In Krötzsch, M., Straccia, U., eds.: RR. Volume 7497 of LNCS., Springer (2012) 106–121
20. Jansen, J., Jorissen, A., Janssens, G.: Compiling input$*$ FO$(\cdot)$ inductive definitions into tabled Prolog rules for IDP^3. TPLP **13**(4-5) (2013) 691–704
21. Järvisalo, M., Oikarinen, E.: Extended ASP tableaux and rule redundancy in normal logic programs. TPLP **8**(5-6) (2008) 691–716
22. Kowalski, R.: Logic for Problem Solving. Volume 7 of The Computer Science Library, Artificial Intelligence Series. North Holland, New York, Oxford (1979)
23. Lloyd, J.W.: Foundations of Logic Programming. Springer-Verlag New York, Inc. (1987)
24. Syrjänen, T.: Lparse 1.0 user's manual. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`
25. Vandevoorde, D., Josuttis, N.: C++ Templates: The Complete Guide. Pearson Education (2002)