Bootstrapping inference in the IDP Knowledge Base System

Bart BOGAERTS and Joachim JANSEN and Broes DE CAT and Gerda JANSSENS and Maurice BRUYNOOGHE and Marc DENECKER

Department of Computer Science, KU Leuven
Celestijnenlaan 200A, 3001 Heverlee, BELGIUM

firstname.lastname@cs.kuleuven.be

Received December 3, 2015

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.

Keywords IDP, Bootstrapping, Declarative Systems, Meta, FO(·)

§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) [21], Constraint Programming (CP) [2], and Logic Programming (LP) [25, 26], including Answer Set Programming (ASP) [3, 17]. 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 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[*1] 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++ [28]. Recently, meta-programming has been applied in ASP. For example, Gebser et al. [19] present a meta-approach to declaratively debug ASP programs. In [22], 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 [5, 7], a Knowledge Base System (KBS) [13] for the logic $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, 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. The language supported by IDP is very rich and leaves users lots of freedom on how to specify their constraints. In order to perform the different types of inference efficiently, often an extensive analysis and several transformations are needed prior to calling the actual in-

[*1] Preliminary version of this paper has appeared in [4]

ference engines. This analysis and transformations ensure that the theories are in a suitable normal form for efficient inference. 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. More concretely, we will show how bootstrapping can be used as a cheap way to implement the aforementioned analysis and transformation tasks. This allows us to reduce the development time of IDP significantly. All of the discussed bootstrapping applications are cast themselves as model expansion tasks, i.e., the task is to find expansions of a given structure over a given finite domain that satisfy a given theory.[*2] However, other inference methods could be used in bootstrapping applications as well.

First, we discuss how to support meta-level representations in $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$. 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

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

This kind of representations is in essence almost equal to the approach taken in the Gödel programming language, where programs are represented by complex terms [20]. Afterwards, we discuss how to extend this approach to full $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$. In this discussion, we research language features that would make the representation 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 meta-domains are always infinite. Indeed, there are infinitely many propositional formulas over a given finite vocabulary; 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 then 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 their subformulas. More concretely, we represent

[*2] See section 2 for a formal definition.

this formula by the following first-order structure:

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

The above structure represents a disjunctive formula (φ_1) with two subformulas (φ_4 and φ_2 respectively). The first of these subformulas 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 [18, 19, 22]. 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 $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ theory.

Afterwards, we show that several tasks in the workflow of our model expansion engine can themselves be cast as model expansion tasks on the meta-level and solved using the same engine. This allows us to effectively *bootstrap* parts of our engine, reduces development effort and yields more flexible, bug-free, and maintainable code. We consider the following tasks:

- detect stratification in a definition, and split the definition accordingly,
- split the theory used for model expansion in separate components that can be handled independently to boost efficiency,
- compute theory transformation, based on a declarative description of desired properties (e.g., flatness) of the result,
- compute well-typed theories that correspond to a partially typed one (i.e., from incomplete input),
- calculate more abstract representations of theories from specific ones.

These tasks are all cast as model expansion tasks. Model expansion is closely related to constraint programming and answer set generation, as discussed in [11]. Hence, each of these 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 com-

pares several solutions to meta-modelling in FO$(\cdot)^{\texttt{IDP}}$, presents a set of new applications of meta-modelling, and shows how these applications 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$(\cdot)^{\texttt{IDP}}$ representations and discuss their properties. In Section 4, we present how the above tasks are tackled using these representations. In Section 5, we describe our implementation of these bootstrapping applications in IDP and present an empirical evaluation of the resulting system. Concluding remarks and future work follow in Section 6.

§2 Preliminaries

In this section we present the language FO$(\cdot)^{\texttt{IDP}}$, focusing on the aspects that are relevant for this paper. Details and examples can be found in [5,7]. We assume familiarity with basic concepts of FO. FO$(\cdot)^{\texttt{IDP}}$ 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_i and d' are domain elements. We use *three-valued* first-order *structures*. Formally, a structure consist of a domain and an assignment of 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$(\cdot)^{\texttt{IDP}}$, 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 $\forall \overline{x} : p(\bar{t}) \leftarrow \varphi$, (or $\forall \overline{x} : f(\bar{t}) = t' \leftarrow \varphi$) where φ is an FO formula and the free variables of φ and $p(\bar{t})$ are among the $\overline{x}$. We call $p(\bar{t})$ (respectively $f(\bar{t}) = t'$) 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 $\forall \overline{x} : p(\bar{t}) \leftarrow \varphi$ is *not* a shorthand for $\forall \overline{x} : p(\bar{t}) \vee \neg\varphi$. 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 $graph_f(\bar{t}, t) \leftarrow \varphi$); 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 a structure I, both over a vocabulary Σ, and a 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

There are several approaches for developing a meta-language to reason about FO$(\cdot)^{\text{IDP}}$, 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. In the rest of this paper, all applications will use 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 3.1
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:

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

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 [9], but this is far from finished. For certain types of problems, these infinite domains will not be problematic. For example, as we discuss in Section 4.1, sometimes model expansion can be reduced to evaluating a Prolog program, where infinite

domains form no problem. However, this will not work for every theory, 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 wish to obtain 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 tree 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 $arity[s : Symbol \to n : Index]$, which expresses that s has arity n, and a partial function $type[s : Symbol, i : Index \to t : Type]$,

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.

The previous meta-symbols all serve to represent vocabularies; in order to also represent FO$(\cdot)^{\texttt{IDP}}$ theories, we add a type *TheoryComponent* to represent all syntactic constructs used to describe theories (formulas, variables, terms, rules, definitions) abbreviated TC in what follows. 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* that is used to distinguish the different kinds of theory components; this type has a fixed interpretation consisting of
  - identifiers for the various types of formulas: *conj* (conjunctive formula), *disj* (disjunctive formula), $quant_{univ}$ (universal quantification), $quant_{exist}$ (existential quantification), *atom* (atom), etcetera,
  - identifiers for various types of terms: *domElem* (domain elements), *var* (variables) or $functerm$ (terms obtained by function application),
  - identifiers for other types of components: *rule* (rules) and *def* (definitions).
- a function $kind[t : TC \to k : Kind]$ that maps every theory component t to the kind of component it represents (for example, the formula $\forall x : P(x)$ would be mapped to $quant_{univ}$),
- partial functions that map components to their constituent subcomponents, such as $symbol[t : TC \to s : Symbol]$, that maps atoms and terms obtained by function application t to their predicate/function symbol s, and $subcomp[t : TC, i : Index \to st : TC]$, which indicates that st is the i'th subcomponent of t (for example a quantified formula has two subcomponents: the variable it quantifies over and one subformula; a conjunction has two subformulas),
- a partial function $typeOf[tc : TC \to t : Type]$ that maps all terms to their type.

Similar types and functions can straightforwardly be defined to also represent structures, but this is not necessary in this paper.

One advantage of this approach is that we can add extra domain elements to TC, 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 sec-

tion, 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 3.2
Let $p[T, T]$ be a predicate symbol and $f[T \to T]$ a function symbol. The information about the symbols in a formula $\forall x[T] : p(x, f(x))$ is represented by

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

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, and vx to represent the term $f(x)$, and the variable x respectively. The remaining information can then be encoded as follows.

$$\begin{aligned}
&\mathit{TC} &&= \{\varphi_1, \varphi_2, t, vx\},\\
&\mathit{kind} &&= \{\varphi_1 \mapsto \mathit{quant}_{univ},\ \varphi_2 \mapsto \mathit{atom},\ t \mapsto \mathit{functerm}, vx \mapsto \mathit{var}\},\\
&\mathit{subcomp} &&= \{(\varphi_1,1) \mapsto vx, (\varphi_1,2) \mapsto \varphi_2,\ (\varphi_2,1) \mapsto vx,\ (\varphi_2,2) \mapsto t,\\
&&&\qquad (t,1) \mapsto vx\},\\
&\mathit{symbol} &&= \{\varphi_2 \mapsto p,\ t \mapsto f\}\\
&\mathit{typeOf} &&= \{t \mapsto T,\ vx \mapsto T\}
\end{aligned}$$

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 detailed way to represent FO$(\cdot)^{\text{IDP}}$ theories. For most applications such a detailed representation is not necessary and an abstraction of the detailed information suffices. Furthermore, often the above described language is too low-level to be practically usable. For example, in Section 4.1 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 that a representation at the "right" level of abstraction is available a priori. This makes our applications much more readable and the knowledge represented in our meta theories much more natural. Of course, what this right level is depends on the task at hand. We will in each of our bootstrapping applications clearly mention this right level of abstraction by providing the vocabulary used by the application. In Section 4.5 we show that obtaining input in the right abstraction level can itself be cast as a bootstrapping task.

§4 Bootstrapping IDP

Consider the model expansion task $MX\,\langle \Sigma, \mathcal{T}, I, \sigma_{out} \rangle$. In this section, we discuss how several subtasks of IDP's model expansion workflow can be cast as model expansion tasks themselves, using a meta-modelling approach. The workflow of the model expansion engine in IDP consists of the following steps (for more information, we refer to [7, 10]):

1. Derive a unique well-typed theory from the partially-typed input theory. If none or multiple exist, an error is produced.
2. Split definitions as much as possible.
3. Split the theory used for model expansion in four parts: (**a**) a part whose models can be computed efficiently in advance, (**b**) a part without special properties, (**c**) a part that can be evaluated in a post-processing step and (**d**) a part that is irrelevant for the task at hand and that can be ignored without changing soundness of the model expansion inference.
4. Compute (**a**).
5. Transform (**b**) into an equivalent theory (**b'**) in a suitable normal form (quantifications and negations pushed inwards, ...).
6. Ground (**b'**).
7. Apply a search algorithm to the ground theory of the previous step.
8. Postprocess (**c**) to complete the (partial) model found in the previous step.
9. Project the model to the output vocabulary.

In this section, we discuss how four of the above steps can be bootstrapped: in Sections 4.1, 4.2, 4.3, and 4.4 we handle steps 3, 2, 5, and 1 respectively. Afterwards, in Section 4.5, we discuss how generating input at the right level of abstraction for the above tasks can itself be performed as a bootstrapping task.

4.1 Splitting the Model Expansion Task

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. [23] have shown 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.

Example 4.1
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 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. This kind of definitions has been called input$*$-definitions by Jansen et. al. [23].
- **Postprocess**: Definitions that can be evaluated after search (so-called output$*$-definitions).
- **Forget**: Definitions that are irrelevant for this model expansion problem.
- **Search**: Definitions without special properties, i.e., that should be con-

sidered during search.

The definitions we preprocess are highly similar to domain predicates in the ASP grounder lparse [27]. 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; in Section 4.5 we discuss how this representation can be obtained):

- types: *Def* and *Symbol* to represent definitions and symbols respectively,
- relations $hasOpen[d : Def, s : Symbol]$ and $defs[d : Def, s : Symbol]$, with intended interpretation that d has s as an open symbol, respectively d defines s,
- a relation $occursInFO[s : Symbol]$, meaning that s occurs in FO sentences of the theory,
- 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

- $depends[s_1 : Symbol, s_2 : Symbol]$ meaning that a definition[*3] in which s_1 is defined depends on s_2 (i.e., has s_2 as an open),
- $outRel[s : Symbol]$ meaning that the value of s is relevant for computing the values of 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-

[*3] It is possible that a predicate symbol is defined in two different definitions. In such case, both definitions are constrained to have the same model for that symbol.

and preprocessing steps as much as possible:

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

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

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 open symbols are either input or defined in another definition that will be preprocessed. 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 [24], 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 (note that these symbols will also not contribute to any form of *extended resolution*, e.g., as described in [6], since these symbols are not used in any of the constraints).

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. Furthermore, determining in which category every definition fits does not require search: evaluating the definitions of *depends*, *pre*, *search*, etcetera can be done very efficiently

during the preprocessing step itself.

4.2 Splitting Definitions

In this Section, we are concerned with splitting large inductive definitions into a partition of several smaller ones. Theoretically, detecting equivalence of the partition with the original definition has already been studied intensively. Below, we repeat a result from [12], and show how this can be reformulated as an FO($\cdot$) theory over a meta-vocabulary, over an abstraction of the vocabulary described in Section 3.2.

Definition 4.1 (Partition)
Let Δ be a definition. A *partition* of Δ is a set $\{\Delta_1, \ldots, \Delta_n\}$ of definitions such that Δ is the disjoint union $\Delta_1 \cup \cdots \cup \Delta_n$, and for defined symbols p, if p is defined in Δ_i, then all rules defining p are in Δ_i.

Example 4.2
Let Δ, Δ_1 and Δ_2 as below:

$$\Delta = \left\{ \begin{array}{l} p \leftarrow q. \\ q \leftarrow r. \end{array} \right\}, \Delta_1 = \left\{ \; p \leftarrow q. \; \right\}, \Delta_2 = \left\{ \; q \leftarrow r. \; \right\}.$$

Then $\{\Delta_1, \Delta_2\}$ forms a partition of Δ, and Δ is equivalent with $\Delta_1 \wedge \Delta_2$.

Example 4.2 shows that sometimes a partition of a definition is equivalent with the original definition. This is not always the case. Not every definition can be split arbitrarily preserving equivalence, as shown in Example 4.3.

Example 4.3
Let Δ, Δ_1 and Δ_2 as below.

$$\Delta = \left\{ \begin{array}{l} p \leftarrow q. \\ q \leftarrow p. \end{array} \right\}, \Delta_1 = \left\{ \; p \leftarrow q. \; \right\}, \Delta_2 = \left\{ \; q \leftarrow p. \; \right\}.$$

Then $\{\Delta_1, \Delta_2\}$ forms a partition of Δ, but Δ is not equivalent with $\Delta_1 \wedge \Delta_2$: $\{p, q\}$ is a model of $\Delta_1 \wedge \Delta_2$, but not of Δ.

Definition 4.2 (Dependency)
Let Δ be a definition and p, q defined symbols in Δ. We say that p *depends directly on* q if q occurs in a rule defining p. We say that p *depends on* q if (p, q)

is in the transitive closure of the "depends directly on" relation.

The following theorem is a reformulation of a result from [12].[*4]

Theorem 4.1
Let Δ be a definition and $\{\Delta_1, \ldots, \Delta_n\}$ a partition of Δ. If for every i and every two symbols p and q defined (in Δ) such that p depends on q and q depends on p, it holds that p is defined in Δ_i if and only if q is defined in Δ_i, then Δ and $\Delta_1 \wedge \cdots \wedge \Delta_n$ are equivalent.

Theorem 4.1 is not only useful from a theoretical point of view, but also practically. In Section 4.1, we already explained that it can sometimes yield large performance gains to pre- or post-process certain definitions during the model expansion task. Sometimes, we cannot use these dedicated techniques to pre- or post-process the entire definition, but only parts of it, as the example below illustrates.

Example 4.4
Let $\mathcal{T}$ be the theory consisting of one definition

$$\Delta = \left\{ \begin{array}{l} p \leftarrow q. \\ q \leftarrow r. \\ t \leftarrow r \wedge s. \end{array} \right\}$$

and one FO sentence

$$q \vee s.$$

Now, if only r is interpreted in the input structure, the methods described in Section 4.1 no longer work: there are no definitions that can be postponed or preprocessed. However, there is a partition of the definition of $\mathcal{T}$ that is equal to the theory from Example 4.1. Hence, splitting that definition ensures that the methods from Section 4.1 yield better results.

There exists a most precise partition of a definition such that the conditions of Theorem 4.1 are satisfied, namely:

[*4] In fact, the result in [12] is more refined than the one presented here: the result here represents a splitting based on a dependency on the level of predicate symbols, while Theorem 5.20 from [12] is a splitting result based on a (more general) dependency on the level of atoms.

Definition 4.3 (Symbol-optimal partition)
Let Δ be a definition. The *symbol-optimal partition* of Δ is a partition $\{\Delta_1, \ldots, \Delta_n\}$ of Δ such that two symbols p and q defined in Δ are in the same Δ_i if and only if p depends on q and vice versa.

The partition in the above definition is called *symbol*-optimal partition because it is the most precise partition of Δ that can be obtained by reasoning on a dependency relation of the defined symbols. Other dependency relations could be used, e.g., by reasoning on domain atoms; this was also done in [12].

We can model the symbol-optimal partition as an FO($\cdot$) theory using an abstraction of the meta-vocabulary defined above. The vocabulary needed to model this consist of:

- three types: *Rule*, *Def* and *Symbol*,
- relations $in[r : Rule, d : Def]$, $occursIn[s : Symbol, r : Rule]$ describing that r is a rule in d and that symbol s occurs in rule r respectively; and a relation $defDefs[d : Def, s : Symbol]$ meaning that definition d defines the symbol s,
- a function $ruleDefs[r : Rule \to s : Symbol]$ meaning that rule r defines the symbol s,
- a constant $D : Def$ that distinguishes Δ from the Δ_i (D refers to Δ, all other definitions are supposed to be among the Δ_i),
- and an auxiliary relation $dep[s_1 : Symbol, s_2 : Symbol]$ with intended meaning that s_1 depends on s_2 (either they are equal or s_1 is defined in terms of s_2 (or in terms of symbols that depend on s_2).

The following theory then describes the unique symbol-optimal partition of Δ.

All definitions other than D form a partition of D:

$$\forall r : in(r, D) \Rightarrow \#\{d \mid d \neq D \wedge in(r, d)\} = 1.$$
$$\forall r, d : in(r, d) \Rightarrow in(r, D).$$

Two rules that define the same symbol are in the same definition:

$$\forall r, r' : ruleDefs(r) = ruleDefs(r') \Rightarrow \forall d : in(r, d) \Leftrightarrow in(r', d).$$

Definition of dependency:

$$\left\{ \begin{array}{l} \forall s, s' : dep(s, s') \leftarrow \exists r : ruleDefs(r) = s \wedge occursIn(s', r). \\ \forall s, s'' : dep(s, s'') \leftarrow \exists s' : dep(s, s') \wedge dep(s', s''). \end{array} \right\}$$

Condition for symbol-optimal partition

$$\forall d, s, s' : d \neq D \wedge defDefs(d, s) \wedge defDefs(d, s') \Rightarrow dep(s, s') \wedge dep(s', s).$$

Relationship between *defDefs* and *ruleDefs*:

$$\forall d, s : defDefs(d, s) \Leftrightarrow \exists r : in(r, d) \wedge ruleDefs(r) = s. \tag{1}$$

Models of Theory 1 correspond exactly to the symbol-optimal partition of Δ. The *input* of the splitting task is a partial structure interpreting the types (*Rule*, *Def*, and *Symbol*), and the constant D. Additionally, it knows the structure of all rules, i.e., it interprets *occursIn* and *ruleDefs*. Furthermore, it contains a partial interpretation of *in*, i.e., it interprets which rules are in Δ and which are not, but it does not contain information about the Δ_i. Contrary to the previous section, this is not purely an analysis task. New symbols (representing the Δ_i) have to be introduced. In order to be sure to have enough new symbols, we need as many new symbols as there are defined symbols in Δ, since this is an upper bound on the number of definitions in the resulting partition. The *output* of the splitting task consists of a two-valued structure over the above vocabulary. The most important is that the predicate $in[Rule, Def]$ is interpreted in this structure. Using this predicate, the output structure can be automatically translated back to a set of definitions, i.e., to the desired partition of the original definition. This approach might generate multiple *symmetrical* solutions by simply permuting the labels assigned to each of the definitions (e.g., $\Delta_1 = \{r_1\}, \Delta_2 = \{r_2\}$ versus $\Delta_2 = \{r_1\}, \Delta_1 = \{r_2\}$). Of course, such symmetrical solutions are not of interest. This kind of behaviour can either be avoided using IDP's built-in symmetry breaking [15, 16] or by a manual encoding of symmetry breaking constraints. In our implementation, we used a manual encoding: we compute the resulting theory in two phases: first, we compute the partitions as a binary relation on

the rules (hence eliminating symmetry) and only afterwards introduce the "new" definition symbols.

In this application, the advantages of bootstrapping are quite prominent. The code is very compact: it contains three lines to define partitions satisfying the conditions in Theorem 4.1, two lines to define dependencies (a generic part of the theory that might be reused in other applications) and one line to enforce symbol-optimal partitions. It is clear that this code is easily maintainable: if a developer wishes to generate partitions based on another criterion than symbol-optimality, she only needs to modify one line of code. If a developer ever wishes to join different definitions rather than split them, even the same theory can be used. Furthermore, it is much easier to see correctness of this theory than correctness of an imperative program used to split definitions. This is because our declarative specification is a direct formalisation of our theoretical understanding, namely of Definitions 4.2 and 4.3 and Theorem 1.

4.3 Transformations

In this section, we present an approach to perform equivalence-preserving transformations on an $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ theory using bootstrapping. We use the encoding that is presented in Section 3.3.

The main idea in this section is that we do not describe a transformation, but instead, we describe (**i**) a set of equivalent formulas and (**ii**) the properties that the final theory should satisfy. In order to define (**i**), we use a relation $appEq[F, F]$ relation. This relation represents an approximation of the equivalence relation between formulas (abbreviated F), given some syntactic equivalence rules. Such a check is always only an approximation, as the equivalence of FO formulas (and, hence, also $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ formulas) is undecidable in general.

Below, we show part of a definition of the *appEq* relation. This definition assumes partial functions $not[F \to F]$, $forall[Var, F \to F]$ similar to the constructors used in Sections 3.1 and 3.2. Of course, these functions are now partial functions since we no longer assume that all formulas are part of the domain.

$$\left\{\begin{array}{l}
\text{Equivalence is reflexive:}\\
\quad \forall \varphi_1 : appEq(\varphi_1, \varphi_1).\\
\text{Equivalence is symmetric:}\\
\quad \forall \varphi_1, \varphi_2 : appEq(\varphi_1, \varphi_2) \leftarrow appEq(\varphi_2, \varphi_1).\\
\text{Equivalence is transitive:}\\
\quad \forall \varphi_1, \varphi_2 : appEq(\varphi_1, \varphi_2) \leftarrow \exists \varphi_3 : appEq(\varphi_1, \varphi_3) \wedge appEq(\varphi_3, \varphi_2).\\
\text{Double negations can be eliminated:}\\
\quad \forall \varphi_1 : appEq(not(not(\varphi_1)), \varphi_1).\\
\text{Negations can be pushed through quantifications:}\\
\quad \forall x, \varphi_1 : appEq(not(forall(x, \varphi_1)), exists(x, not(\varphi_1))).\\
\text{Subformulas can be replaced by equivalent subformulas:}\\
\quad \forall x, \varphi_1, \varphi_2 : appEq(forall(x, \varphi_1), forall(x, \varphi_2)) \leftarrow appEq(\varphi_1, \varphi_2).\\
\ldots
\end{array}\right\}$$

The properties the resulting theory should satisfy are often easily expressible in FO. For example, for a transformation that transforms theories into negation normal form this would be:

$$\begin{array}{l}
\text{The output formula is equivalent with the input:}\\
\quad appEq(\varphi_{input}, \varphi_{output}).\\
\text{Negations only occur in front of atoms in the output formula:}\\
\quad \forall \varphi : subfOf(\varphi, \varphi_{output}) \wedge kind(\varphi) = neg \Rightarrow kind(subform(\varphi, 1)) = atom
\end{array}$$

This reduces the transformation to a model expansion task: find an interpretation for φ_{output} such that all constraints hold. This is highly reusable code because for a new transformation only this second part has to be rewritten to contain the new desired properties. The first part can be reused in its entirety or extended. Furthermore, this approach also allows us to perform several transformations at once. For example, in order to transform a propositional formula to CNF one would first push negations inward, and next apply the distributivity as much as possible. With this approach on the other hand, the control over these processes is handed to the solver, which can decide to merge them or handle them as separate processes. One now simply states that the resulting formula should be in CNF and all control over how this form is obtained is a burden the user is freed from. Furthermore, this approach has as advantage that one can search for models that are optimal with respect to certain criterion. For example, the task can be extended to finding a theory, equivalent with the

original one such that the theory is in negation normal form, flattened and such that the size of the grounding of this theory is minimal (given an interpretation of the types).

When using this approach in a finite domain solver, one should carefully overestimate the number of extra introduced "spare" domain elements. A model of this theory should not only have enough domain elements to represent the result of the transformation, but also to represent all intermediary steps in the derivation that this formulas is equivalent to the original one. For example, in order to transform $not(not(not(not(not(not(f))))))$ to NNF with the above definition of *appEq*, we need at least 2 extra domain elements, namely to represent $not(not(not(not(f))))$ and $not(not(f))$.

4.4 Type Derivation

For the sake of user-friendliness, many programming and modelling languages do not require that the user specifies all types used in a program or theory. Instead, they try to automatically derive the intended type. This also holds for IDP. For example, if a vocabulary Σ declares the types *Human* and $\mathbb{N}$ and the function $height[Human \to \mathbb{N}]$, then the formula

$$\exists h : height(h) = 170 \tag{2}$$

is unambiguous, even though h is untyped. Indeed, as the vocabulary defines the signature of *height*, one can derive that the user actually means h to range over all humans:

$$\exists h[Human] : height(h) = 170.$$

However, if Σ also contains a type *Vehicle* and a function $height[Vehicle \to \mathbb{N}]$, Formula (2) is ambiguous. It is no longer clear which "*height*" function is referred to and hence what the type of h must be.

In this section, we show how to partially typed theories can be disambiguated using a bootstrapping approach. In IDP, ambiguity is possible because types are order-sorted, i.e., they can have sub- and supertypes, and because overloading of predicate/function symbols is possible, i.e., symbols with different type signatures can have the same name. For ease of presentation, we restrict ourselves to the latter case by only considering many-sorted logic, for which the first two causes of ambiguity cannot occur.

The input to the task is (an encoding of) a partially typed theory: a theory in which the type of some variables or the symbol associated with an atom

or function term is unknown. In the latter case, we will always know the *name* of the symbol, but not yet which specific overloaded symbol was meant. The output should either be a fully typed theory, or an error if no or multiple corresponding fully typed theories exist that correspond with the partially typed one. In order to model this as a meta-level application, we use the encoding introduced in Section 3.3, complemented with:

- A type *Name* that contains the names of all symbols,
- a partial functions $symbolName[TC \to Name]$ that maps function terms and atoms to the name of their symbol,
- a function $name[Symbol \to Name]$ that maps symbols to their name.

The following theory then describes what properties a theory should satisfy to be well-typed. In this theory we assumed that a coreference analysis was executed before, i.e., that it is known which occurrence of variables refer to the same variable and that these occurrences all use the same domain element of type *var*. For example a formula $(\forall x : P(x)) \vee (\forall x : Q(x))$ should be transformed to a structure with two "*var*" objects, one for each quantification over x.

All function terms and atoms have a symbol and a symbolname:

$\forall t : (\exists s : symbol(t) = s) \Leftrightarrow (kind(t) = functerm \vee kind(t) = atom).$

$\forall t : (\exists n : symbolName(t) = n) \Leftrightarrow (kind(t) = functerm \vee kind(t) = atom).$

SymbolName should correspond to the names of the assigned symbols:

$\forall t : (kind(t) = functerm \vee kind(t) = atom)$
$\Rightarrow name(symbol(t)) = symbolName(t).$

If a term has a subterm, then the type of the subterm should be the one expected at that location (type correctness):

$\forall t, i, t_{sub} : (kind(t) = functerm \vee kind(t) = atom) \wedge subcomp(t, i) = t_{sub}$
$\Rightarrow type(symbol(t), i) = typeOf(t_{sub}).$

The type of a function term is its output type:

$\forall t : kind(t) = functerm \Rightarrow typeOf(t) = type(symbol(t), arity(symbol(t)) + 1).$

A partially typed theory then corresponds to a partial structure in the above vocabulary. A partially typed theory is unambiguous, if the above theory has exactly one model that expands the corresponding partial structure.

Example 4.5

Let us consider Formula (2) again, over the extended vocabulary. The input structure then interprets $symbolName(t_{height(h)})$ as *height* and $name(height[Human \to \mathbb{N}])$ and $name(height[Vehicle \to \mathbb{N}])$ both as *height*. The above theory will then

have two models: one interprets $symbolName(t_{height(h)})$ as $height[Human \to \mathbb{N}]$ and one as $height[Vehicle \to \mathbb{N}]$. Hence, no unique fully typed theory exists.

4.5 Marshalling

In Section 3.4 we mentioned that an abstraction of the most detailed specification suffices for many applications. Such abstraction can themselves be obtained through bootstrapping. For example, in Section 4.1 we described an application where the model expansion task is split into several subtasks given an input theory. The only input this application actually requires is:

- 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 are defined in which definition is done with

Definition d defines symbol s if for some rule r in d,
the symbol defined in the head (first component) of r is s.

$$\left\{ \begin{array}{l} \forall s, d : defs(d, s) \leftarrow kind(d) = def \wedge \exists i, r : subcomp(d, i) = r \\ \qquad\qquad\qquad\qquad\qquad \wedge symbol(subcomp(r, 1)) = s. \end{array} \right\}$$

Given the most detailed representation of the theory and a definition that describes the abstraction, definition evaluation yields the desired abstraction.

4.6 More bootstrapping applications

The applications discussed above in details are only the tip of the iceberg. 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 [9].

Furthermore, in this paper we focused on applications that could be cast as model expansion tasks. 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 [8].

§5 Discussion & Evaluation

In the previous sections, we presented several ideas concerning meta-level representations of $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ and bootstrapping applications that use these meta-level representations. We implemented several of these ideas in the IDP system. More concretely, the ideas presented in Sections 4.1 and 4.2 are now by default part of IDP's model expansion workflow, implemented using bootstrapping.[*5] Of course, 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. We also implemented a general transformation of internal (C++) data-structures representing an $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ theory to a highly general meta-representation as described in Section 3.3 and back.[*6] These transformations allow us to implement the other bootstrapping ideas we presented in Section 4. However, as discussed in Section 4.3, each transformation we implement will require us to overestimate the number of additional elements needed.

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 definitions are already split as much as possible, or that the theory does not need to be partitioned in the four components described in Section 4.1. We make our own theories as such that they require none of the various bootstrapping methods themselves. Thus, when a user performs a model expansion task, internally we will start new model expansion tasks for the bootstraps part of the inference. However, in *those* model expansion tasks, no bootstrapping will occur (or be needed). As such, we avoid the pitfall of an infinite nesting of model expansion calls.

5.1 Evaluation

In this section we evaluate our bootstrapping implementation of *splitting definitions* and *pre- and post-processing definitions*. We answer the following questions. **(Q1)** How big is the overhead of bootstrapping in case nothing can be

[*5] The entire IDP code used for these applications can be found in any release of IDP (version 3.3.1 and above) in the file `share/std/definitions.idp`.

[*6] The procedures for transforming from and to meta representations are available in any release of IDP (version 3.3.2 and above) and are called `tometa` and `frommeta` respectively.

gained from these techniques? **(Q2)** How big are the possible gains of applying these techniques.

All our experiments our ran on a machine with an Intel©Xeon(R) CPU E31230 @ 3.20GHz with 8Gb RAM with a time limit of one hour and a memory limit of 4Gb.

To evaluate **(Q1)**, we ran two version of IDP on a set of IDP-encodings of problems from the ASP competitions.[*7] The first version of IDP, simply referred to as IDP below has all bootstrapping options disabled; the second version, referred to as IDP-BOOT has bootstrapping enabled. For each of the problems, we ran both versions of IDP on 5 randomly selected instances (note that the bootstrapping applications we discuss do not depend on the instance, since they are theory transformations). We report only on the grounding times for both versions; since there is no difference in the solving process. Average grounding times (averaged over 5 instances for each problem) of both versions are listed in Table 1.

Problem	**IDP**	**IDP-boot**	**Difference (stdev)**
Bottle Filling	3.582	**3.430**	0.152 (0.016)
Graceful Graphs	**0.217**	0.259	-0.042 (0.015)
Hamiltonian Path	**0.084**	0.195	-0.111 (0.007)
N-Queens	164.106	**163.848**	0.258 (0.967)
Permutation Pattern Matching	**0.722**	0.849	-0.126 (0.016)
Solitaire	**0.255**	0.491	-0.236 (0.007)
Stable Marriage	**71.039**	71.454	-0.415 (0.511)

Table 1: Average grounding times, grounding time difference and standard deviation of the grounding time difference (in seconds) of IDP and IDP-BOOT on the different problem domains. Positive difference mean IDP-BOOT was faster; negative difference means IDP was faster.

Our finding is that whenever the grounding is non-trivial (when it cannot be computed in a fraction of a second), the difference in grounding time is negligible, and hence that for these theories, enabling bootstrapping does not result in a significant overhead.

Regarding **(Q2)**, it deserves to be stressed that these bootstrapping

[*7] The encodings we used are available at `https://bitbucket.org/krr/benchie/src/6eb1e13220470efca078fde58b167a978c3c2df0/idp_speedtest/Problems/ModelExpand/?at=master`

techniques will result in zero gain on highly optimised theories, e.g., such as the ones that are sent to competitions. Indeed, these theories are developed by experts who know the inner working of the system very well; such experts will manually split their definitions and avoid writing definitions that can be ignored anyway. Instead, our optimisations are aimed towards non-expert users who rely on automated optimisations. Hence, in order to evaluate **(Q2)**, we compared IDP and IDP-BOOT on files containing non-optimal encodings. To be precise, we compared the performance of IDP and IDP-BOOT on the following encoding of the N-queens problem while varying the size of the board. We use a vocabulary containing

- types: *index* to represent indices of rows and columns and *diagtype* to represent indices of diagonals,
- relations $queen[x : index, y : index]$ to represent that there is a queen on position (x, y) and $attacked[x : index, y : index]$ to represent that position (x, y) is attacked by some queen,
- functions $diag1[x : index, y : index \to d : diagtype]$ and $diag2[x : index, y : index \to d : diagtype]$ that map each position (x, y) to some d such that (x, y) lies on the d'th main diagonal, respectively the d'th side diagonal.

Our model contains the following definitions and constraints:

There can be no queen on a position that is attacked:

$$\forall x, y : \mathit{attacked}(x, y) \Rightarrow \neg \mathit{queen}(x, y).$$

Definition of attack, and the diagonals:

$$\left\{\begin{array}{l}
\text{Computation of the diagonals:} \\
\forall x, y : \mathit{diag1}(x, y) = x - y + \mathit{MAX}[: \mathit{index}]. \\
\forall x, y : \mathit{diag2}(x, y) = x + y - 1. \\
\text{A position is attacked if there is a queen on the same row:} \\
\forall x, y : \mathit{attacked}(x, y) \leftarrow \mathit{queen}(x', y) \wedge x \neq x'. \\
\text{A position is attacked if there is a queen on the same column:} \\
\forall x, y : \mathit{attacked}(x, y) \leftarrow \mathit{queen}(x, y') \wedge y \neq y'. \\
\text{A position is attacked if there is a queen on the same diagonal:} \\
\forall x, y : \mathit{attacked}(x, y) \leftarrow \mathit{queen}(x', y') \wedge \mathit{diag1}(x, y) = \mathit{diag1}(x', y') \\
\qquad\qquad \wedge (x \neq x' \vee y \neq y'). \\
\forall x, y : \mathit{attacked}(x, y) \leftarrow \mathit{queen}(x', y') \wedge \mathit{diag2}(x, y) = \mathit{diag2}(x', y') \\
\qquad\qquad \wedge (x \neq x' \vee y \neq y').
\end{array}\right\}$$

Figure 1 contains a graph with runtimes of IDP and IDP-BOOT for increasing board sizes. As can be seen, on the smallest (almost trivial) instances there is a small overhead (0.15 seconds) caused by the bootstrapping. However, IDP-BOOT is able to handle much larger instances than IDP. This can be explained by the fact that the definition of *diag1* and *diag2* is in fact an input$*$ definition. However, since this definition is joined with the definition of *attacked*, IDP does not detect this case and cannot deploy its optimisations for input$*$ definitions. IDP-BOOT splits this definition and hence discovers the underlying structure. These findings illustrate *(1)* the importance of preprocessing input$*$ definitions (this is in line with the findings from [23]) and *(2)* that IDP-BOOT's possibility to split definitions can result in an increased number of input$*$ definitions and hence, can help users who are not aware of the possible speed-ups obtained by splitting definitions.

§6 Conclusion

Declarative systems solve complex reasoning tasks over a general input.

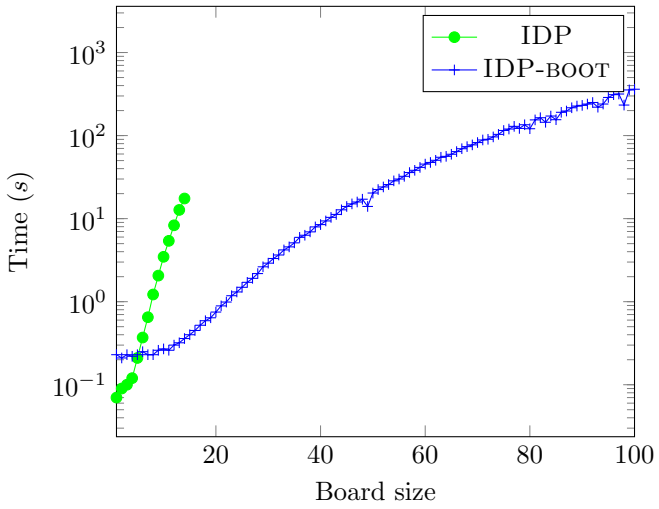

Figure 1: Comparison of run-time of IDP and IDP-BOOT on various instances of the N-queens problem. The x-axis refers to board size (N), while the y-axis refers to runtime in seconds. For boards with size over 14, IDP was not able to find a solution using the provided memory limit of 4Gb.

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 often also be solved declaratively. We presented meta-modelling approaches for $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ to tackle such tasks and diverse applications that are 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) Apt, K.R.: Principles of Constraint Programming. Cambridge University Press (2003)

3) Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, New York, NY, USA (2003)

4) Bogaerts, B., Jansen, J., De Cat, B., Janssens, G., Bruynooghe, M., Denecker, M.: Meta-level representations in the IDP knowledge base system: Towards bootstrapping inference engine development. In Mitchell, D., Denecker, M., eds.: Workshop on Logic and Search, 2014. (2014) 1–14

5) 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 **15** (November 2015) 783–817

6) Chu, G., Stuckey, P.J.: Structure based extended resolution for constraint programming. CoRR **abs/1306.4418** (2013)

7) De Cat, B., Bogaerts, B., Bruynooghe, M., Denecker, M.: Predicate logic as a modelling language: The IDP system. CoRR **abs/1401.6312** (2014)

8) De Cat, B., Bruynooghe, M.: Detection and exploitation of functional dependencies for model generation. TPLP **13**(4-5) (2013) 471–485

9) De Cat, B., Denecker, M., Bruynooghe, M., Stuckey, P.J.: Lazy model expansion: Interleaving grounding with search. J. Artif. Intell. Res. (JAIR) **52** (2015) 235–286

10) De Cat, B., Jansen, J., Janssens, G.: IDP3: Combining symbolic and ground reasoning for model generation. In: 2nd Workshop on Grounding and Transformations for Theories With Variables. (2013) 17–24

11) Denecker, M., Lierler, Y., Truszczyński, 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 Baral, C., De Giacomo, G., Eiter, T., eds.: KR, AAAI Press (2014) 1–10

15) Devriendt, J., Bogaerts, B., Bruynooghe, M.: BreakIDGlucose: On the importance of row symmetry. In: Proceedings of the Fourth International Workshop on the Cross-Fertilization Between CSP and SAT (CSPSAT). (2014)

16) Devriendt, J., Bogaerts, B., De Cat, B., Denecker, M., Mears, C.: Symmetry propagation: Improved dynamic symmetry breaking in SAT. In: IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012, IEEE Computer Society (2012) 49–56

17) 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)

18) Gebser, M., Kaminski, R., Schaub, T.: Complex optimization in Answer Set Programming. TPLP **11**(4-5) (2011) 821–839

19) 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

20) Hill, P.M., Lloyd, J.W.: The Gödel programming language. MIT Press (1994)

21) Hudak, P.: Conception, evolution, and application of functional programming languages. ACM Comput. Surv. **21**(3) (1989) 359–411

22) 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

23) Jansen, J., Jorissen, A., Janssens, G.: Compiling input∗ FO(·) inductive definitions into tabled Prolog rules for idp3. TPLP **13**(4-5) (2013) 691–704

24) Järvisalo, M., Oikarinen, E.: Extended ASP tableaux and rule redundancy in normal logic programs. TPLP **8**(5-6) (2008) 691–716

25) Kowalski, R.: Logic for Problem Solving. Volume 7 of The Computer Science Library, Artificial Intelligence Series. North Holland, New York, Oxford (1979)

26) Lloyd, J.W.: Foundations of Logic Programming. Springer-Verlag New York, Inc. (1987)

27) Syrjänen, T.: Lparse 1.0 user's manual. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`

28) Vandevoorde, D., Josuttis, N.: C++ Templates: The Complete Guide. Pearson Education (2002)