# 1 Predicate Logic as a Modeling Language: The IDP System

AUTHORS: Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, Gerda Janssens, Marc Denecker

Department of Computer Science, KU Leuven

## 1.1 Introduction

Since the early days of artificial intelligence, it is believed that logic could bring important benefits in solving computational problems and tasks compared to standard programming languages. Kowalski's seminal paper *Predicate Logic as a Programming Language* [Kowalski 1974] was a major step in this direction and laid the foundations for the field of logic programming. It introduced two fundamental ideas: on the declarative level, the use of the Horn clause logic fragment of classical logic; on the procedural level, a procedural interpretation of this logic which made it possible to write algorithms in the formalism. With the technology of the time, Kowalski's paper was a breakthrough for the use of logic in computer science.

Since then, logic programming has fanned out in many directions, but in most extensions and variants, the original key ideas are still present in stronger or weaker form: the use of a rule-based formalism, and the presence of a procedural interpretation. Or at least, if programs are not procedures, they are representations of computational problems, as in Datalog and in Answer Set programming.

In this chapter, we present the IDP system (and language) that, although it builds on the accomplishments of logic programming and contains language constructs based on logic programming, embodies a more pure view of logic as a modeling language. In this view, a logic theory is not a program, it cannot be executed or run; it does not describe an algorithm. A theory, in principle, is not even a specification of a problem. A theory is a bag of information, a description of possible states of affairs of the domain of discourse, a representation of knowledge, or in other words a modeling of the domain of discourse. As such, this view breaks the link that Kowalski had laid between logic and programming. On the other hand, the IDP language contains a language construct, namely inductive definition, that directly descends

| | |
|---|---|
| `Human(John).` | `John is a Human` |
| `Human(Jane).` | `Jane is a Human` |
| `Male(John).` | `John is a Male` |
| `Female(x) :- Human(x), notMale(x).` | `Females are Humans that are not Male` |
| `?- Female(Jane).` | `is Jane Female?` |

*yes*

**Table 1.1** Prolog answers "yes" to the query `?- Female(Jane).`

from logic programming, and the IDP system is designed to *use* declarative information to solve problems and uses many technologies that were developed in logic programming. As such, the IDP language and system preserve some of the contributions of logic programming but break with some of the fundamental ideas of logic programming.

To explain the IDP language, we need to go back to the early days of logic programming when negation as failure was introduced. On the one hand, conclusions obtained with the negation as failure inference rule were often natural and as desired. This is illustrated by the program and query in Table 1.1. On the other hand, these answers were logically unsound if rules were interpreted as material implications[1]. This problem disturbed the logic programming community for more than a decade and led to the development of stable and well-founded semantics [Gelfond and Lifschitz 1988, Van Gelder et al. 1991]. However, a formal semantics still does not explain the intuition that humans attach to such programs. For that, we need to study the informal semantics of the logic. So far, two informal semantics have been proposed that can explain the intuitive meaning of logic programs. One is the view of logic programs under stable semantics [Gelfond and Lifschitz 1988] as a non-monotonic logic closely related to default logic and autoepistemic reasoning developed by Gelfond and Lifschitz [Gelfond and Lifschitz 1991]. The second is the view of logic programs as **definitions** of concepts. This view was implicit already in Clark's work [Clark 1978] on completion semantics and in the work by Van Gelder, Ross and Schlipf [Van Gelder 1993, Van Gelder et al. 1991] on the well-founded semantics. It was elaborated later in a series of papers [Denecker 1998, Denecker and Vennekens 2014, Denecker et al. 2001].

The informal concept of definition (as found in scientific texts) has several interesting aspects. First, it is a rule-based linguistic concept of informal language: definitions, certainly inductive ones, are commonly phrased as conditionals or sets of these. Second, definitions are second nature to us. Much human knowledge is of definitional nature; this includes inductive

[1] This is, the logical implication $\varphi \Rightarrow \psi$ that is interpreted as $\neg\varphi \vee \psi$.

and recursive definitions[2]. Many familiar recursive logic programs are obviously representations of inductive definitions (e.g., the `member`, `append` and transitive closure programs). Third, definitions are of mathematical precision: they are the building blocks of formal mathematical theories. Fourth, it is a well-known consequence of the compactness theorem that definitions, in particular inductive ones, cannot, in general, be correctly expressed in classical first-order logic. Fifth, recently [Denecker and Vennekens 2014], it was shown that rule sets under two-valued well-founded semantics correctly formalize all main sorts of informal definitions that we find in scientific text in the sense that the interpretation of the defined symbols in the well-founded model of a rule set always coincides with the set defined by the informal definition represented by the rule set. All these are solid arguments that the concept of definition is a good candidate for the informal semantics of logic programming. Furthermore, they suggest to define a rule-based logic construct under the well-founded semantics for expressing definitions. Importantly, given that this form of information cannot be expressed in classical logic, it makes sense to add such a construct to classical logic. This idea was carried out for the first time in [Denecker 2000] leading to the logic FO(ID) which forms the basis of the IDP language. Given that logic programs were originally seen as a fragment of classical logic, the definition of this logic was certainly a remarkable turn of events.

A definition is a piece of information. It lays a strict, precise, deterministic logical relationship between the defined concept and the concepts in terms of which it is defined. E.g., the definition of transitive closure of a graph specifies a logical relationship between the transitive closure relation and the graph. A definition, like all other language constructs in IDP theories, is not a procedure, it cannot be run, it does not specify a problem. It is a kind of declarative information.

The following question now naturally arises: if IDP theories are not programs, how can they be used to solve computational problems? The IDP system, which supports the IDP language, is conceived as a *knowledge base system (KBS)* [Denecker and Vennekens 2008]. The scientific working hypothesis underlying the knowledge base paradigm is that many computational problems can be solved by applying some form of inference to a specification of information of the problem domain. A KBS essentially consists of two parts. On the one hand, a formal declarative knowledge representation *language* and, on the other hand, a number of powerful and generic *inference methods* to solve a broad class of tasks using a knowledge base.

The paradigm is inspired by several observations. First, *imperative programming* languages allow the direct encoding of specialised algorithms, but knowledge about the problem domain is hidden deep within those algorithms. This facilitates high-performance solutions, but makes debugging and maintenance very difficult. Second, a program is typically written to *perform one task* and perform it well, but cannot handle many related tasks based on the same

[2] In this text, we use the names *recursive definition* and *inductive definition* interchangeably.

knowledge. Third, *knowledge representation* languages excel at representing knowledge in a natural, human-understandable format. Programming language designers are starting to realize this and provide constructs to express generic knowledge, such as the LINQ [Pialorsi and Russo 2007] data queries in C# and annotation-driven configuration [Deinum et al. 2014]. Lastly, the above-mentioned progress in automated reasoning techniques facilitates the shift of the control burden from programmer to inference engine ever more. The knowledge base paradigm is an answer to these observations: application knowledge is modelled in a high-level Knowledge Representation (KR) language and state-of-the-art inference methods are applied to reason on the modelled knowledge. It has also been demonstrated that, while the KBS approach cannot yet compete with highly tuned algorithms, the effort to reach an acceptable solution (w.r.t. computing time or solution optimality) can be much smaller than that to develop an algorithmic solution [Bruynooghe et al. 2015, Gebser et al. 2012a]. Furthermore, the declarative approach results in software that is less error-prone and more maintainable.

The IDP system is a state-of-the art KBS. The system is already in existence for several years, but only recently evolved into a KBS. Up until 2012, IDP was a model expansion system (the IDP2 system)[3] capable of representing knowledge in a rich extension of first-order logic (FO) and performing model expansion by applying its grounder GIDL and its solver MINISAT(ID). Recently, we have extended it into *the IDP knowledge base framework* for general knowledge representation and reasoning (referred to as IDP3); the earlier technology is reused for its model expansion inference. The IDP system goes beyond the KBS paradigm: for a KBS to be truly applicable in practical software engineering domains, it needs to provide an imperative programming interface, see [De Pooter et al. 2011]. Such an interface, in which logical components are first-class citizens, allows users to handle input and output (e.g., to a physical database), to modify logical objects in a procedural way and to combine multiple inference methods to solve more involved tasks. In this chapter, we use KBS to refer to a three-tier architecture consisting of language, inference methods, and procedural integration. The IDP system provides such a procedural integration through the scripting language Lua [Ierusalimschy et al. 1996]. The system's name IDP, *Imperative-Declarative Programming*, also refers to this paradigm.

In the work revolving around IDP, we can distinguish between the knowledge representation language and the state-of-the-art inference engines. One can *naturally* model diverse application domains in the IDP language; this contrasts with many approaches that *encode* knowledge such that a specific inference task becomes efficient. Furthermore, *reuse* of knowledge is central. The IDP language is modular and provides fine-grained management of logic components, e.g., it supports *namespaces*. The implementation of the inference engines provided by IDP aims at the reuse of similar functionality (see Section 1.6). This has two impor-

[3] Given a logical theory and a structure interpreting the domain of discourse, model expansion searches for a model of the theory that extends the structure.

tant advantages: (i) improvement of one inference engine (e.g., due to progress in one field of research) immediately has a beneficial effect on other engines; (ii) once "generic" functionality is available, it becomes easy to add new inference engines. To lower the bar for modellers, we aim at reducing the importance of clever modeling on the performance of the inference engines. Several techniques, such as grounding with bounds [Wittocx et al. 2010], function detection [De Cat and Bruynooghe 2013], automated translation to the most suitable solving paradigm [De Cat et al. 2013] and automated symmetry breaking [Devriendt et al. 2012] have been devised to reduce the need for an expert modeller.

The rest of the chapter is structured as follows. In Section 1.2, we present the syntax and semantics of FO(ID,AGG,PF,T), the logic underlying the system. In Section 1.3 we present a high-level overview of the IDP system. In Section 1.4, we present the IDP language, a user-friendly syntax for FO(ID,AGG,PF,T) language components and procedural control. We present advanced language features and inference methods in Section 1.5. In Section 1.6, we focus on the inner working of some components of the IDP system. More specifically, we describe the workflow of the optimization inference and how users can control the various parts of the optimization engine. Applications, tools and performance are discussed in Section 1.7, followed by related work and a conclusion. In the rest of the chapter, we use IDP to refer to the current (2018), knowledge base version of the system.

This chapter is a tribute to David S. Warren. The XSB Prolog system [Chen and Warren 1996] by David S. Warren and his students was the first to support the well-founded semantics and was a milestone in closing the gap between the procedural semantics of the SLDNF proof procedure [Lloyd 1987] and the intuitive declarative semantics of logic programs as formalized by the well-founded semantics. In fact, XSB is used internally in the IDP system. In a personal communication, David once told the authors of this chapter that when he learned about FO(ID) and the IDP system, he was less than thrilled; specifically, he found it "a crazy idea". It is with great satisfaction and gratitude that we have noticed that he changed his mind as can be seen in his LinkedIn editorial that is devoted to the IDP language [Warren 2014]. It is therefore a great pleasure to contribute this chapter in a book that was initiated to honour his 65th birthday.

## 1.2 FO(ID,AGG,PF,T), the Formal Base Language

In this section, we introduce the logic that is the basis of the IDP language. This logic, FO(ID, AGG,PF,T), is an extension of first-order logic (FO) with inductive definitions, aggregates, partial functions and types.

### 1.2.1 First-Order Logic

A *vocabulary* $\Sigma$ consists of a set of predicate and function symbols, each with an associated *arity*, the number of arguments they take. We sometimes use $P/n$ ($f/n$) to denote the predicate symbol $P$ (function symbol $f$) with arity $n$.

A *term* is a variable or an $n$-ary function symbol applied to $n$ terms. An *atom* is an $n$-ary predicate symbol applied to $n$ terms. An atom is a *formula*; if $\varphi$ and $\varphi'$ are formulas and $x$ is a variable, then $\neg\varphi$, $\varphi\wedge\varphi'$, $\varphi\vee\varphi'$, $\forall x:\varphi$ and $\exists x:\varphi$ are also formulas. The expressions $\varphi\Rightarrow\varphi'$, $\varphi\Leftarrow\varphi'$ and $\varphi\Leftrightarrow\varphi'$ are (as usual) shorthands for $\neg\varphi\vee\varphi'$, $\varphi\vee\neg\varphi'$ and $(\neg\varphi\vee\varphi')\wedge(\varphi\vee\neg\varphi')$ respectively. A *literal* (often denoted $l$) is an atom $a$ or its negation $\neg a$. A *sentence* is a formula without free (unquantified) variables. A *theory* $\mathcal{T}$ over a vocabulary $\Sigma$ consists of a set of sentences with symbols in $\Sigma$. A term $t$ containing occurrences of a term $t'$ is denoted as $t[t']$; the replacement of $t'$ in $t$ by $t''$ is denoted as $t[t'/t'']$ (similarly for formulas).

A *two-valued structure* (in the literature, sometimes also referred to as an *interpretation*) $I$ over a vocabulary $\Sigma$ consists of a *domain* $D$ and an interpretation for all symbols in $\Sigma$; we use $s^I$ to refer to the interpretation of a symbol $s$ in $I$. A two-valued interpretation $P^I$ of a predicate symbol $P/n$ is a subset of $D^n$; a two-valued interpretation $f^I$ for a function symbol $f/n$ is a mapping $D^n\rightarrow D$. The latter mapping can also be represented by a subset of $D^{n+1}$ in which there is a functional dependency from the first $n$ arguments to the last one. Given a domain $D$, a *domain atom* is a tuple $(P,\overline{d})$ where $P$ is an $n$-ary predicate symbol and $\overline{d}\in D^n$ is an $n$-tuple of domain elements. Sometimes, we abuse notation and write a domain atom as $P(\overline{d})$.

While the domain of standard FO is unordered, it is often convenient to assume that there is a total order of the set of all domain elements and that a vocabulary includes, by default, the binary comparison predicates $=/2$, $\neq/2$, $</2$, $>/2$, $\geq/2$ and $\leq/2$; their interpretation is fixed in accordance with the total order.

By evaluating a term or formula in a structure $I$, we obtain its *value*. The value of a term is a domain element, the value of a formula is a truth value, either true, denoted $\mathbf{t}$, or false, denoted $\mathbf{f}$, hence an element of the set $\{\mathbf{t},\mathbf{f}\}$. The value of a term $t$, denoted as $t^I$, is $d$ if $t$ is of the form $f(\bar{t})$ and $f^I(\bar{t}^I)=d$. The value $P(\bar{t})^I$ of an atom $P(\bar{t})$ in $I$ is $\mathbf{t}$ if $\bar{t}^I\in P^I$ and $\mathbf{f}$ otherwise. We define $(\varphi\wedge\varphi')^I=\mathbf{t}$ if $\varphi^I=\varphi'^I=\mathbf{t}$, $\varphi\vee\varphi'^I=\mathbf{t}$ if either $\varphi^I=\mathbf{t}$ or $\varphi'^I=\mathbf{t}$, $\neg\varphi^I=\mathbf{t}$ if $\varphi^I=\mathbf{f}$; $(\forall x:\varphi)^I=\mathbf{t}$ if $(\varphi[x/d])^I=\mathbf{t}$ for all $d\in D$, $(\exists x:\varphi)^I=\mathbf{t}$ if $(\varphi[x/d])^I=\mathbf{t}$ for at least one $d\in D$. In the two quantified forms, the replacement of $x$ by $d$ in $\varphi$ means that $x$ is interpreted as $d$ when deriving the value of $\varphi$ in $I$. We say a two-valued structure $I$ is a *model* of a formula $\varphi$ or $I$ *satisfies* $\varphi$, denoted as $I\models\varphi$, if $\varphi^I=\mathbf{t}$. Given two tuples $\bar{t}=(t_1,\ldots,t_n)$ and $\bar{t}'=(t'_1,\ldots,t'_n)$ of terms of equal length $n$, $\bar{t}=\bar{t}'$ denotes the conjunction $t_1=t'_1\wedge\cdots\wedge t_n=t'_n$. For vocabularies $\Sigma$ and $\Sigma$' with $\Sigma'\supseteq\Sigma$ and a structure $I$ over $\Sigma'$, $I|_\Sigma$ denotes the restriction of $I$ to the symbols in $\Sigma$.

Unless the context specifies it differently, $\varphi$ denotes a formula, $t$ a term, $D$ a domain, $I$ a two-valued structure, $\mathcal{I}$ a partial structure (introduced below), $d$ a domain element, $x$ and $y$ variables, and $\sim$ any comparison predicate.

Sometimes, it is convenient to use *true* and *false* as atoms in a formula. Therefore, we include them as nullary predicates in every vocabulary. In every structure $I$, *true* is interpreted as $\{()\}$, i.e., as the set containing only the empty tuple, hence $true^I=\mathbf{t}$, and *false* is interpreted as the empty set $\emptyset$, i.e., $false^I=\mathbf{f}$.

***Partial Structures*** A typical problem solving setting is that of model expansion [Mitchell and Ternovska 2005] where one has partial knowledge about a structure and where the goal is to expand this partial information into a structure that is a model of the given theory. Hence we also use partial structures. A partial structure over a vocabulary $\Sigma$ consists of a *domain* $D$, and a partial interpretation $I$ of the symbols in $\Sigma$. With $s^I$, we refer to the interpretation of a symbol $s$ in a partial structure $I$.

Whereas the two-valued interpretation of a predicate $P/n$ was defined as a subset of $D^n$, it can as well be defined as a mapping from $D^n$ to the set $\{\mathbf{t},\mathbf{f}\}$. The latter form is better suited for generalization. The partial interpretation of a predicate $P/n$ is defined as a mapping from $D^n$ to the set $\{\mathbf{t},\mathbf{f},\mathbf{u}\}$, with $\mathbf{u}$ standing for "unknown". This mapping partitions $D^n$ in the set of true tuples, denoted $P^I_{ct}$ (here $ct$ stands for *certainly true*), the set of false tuples, denoted $P^I_{cf}$ ($cf$ stands for *certainly false*), and the set of unknown tuples, denoted $P^I_u$. Note that two of these sets fully determine the partial interpretation of $P$.

Whereas the two-valued interpretation of a term is a single domain element, this is no longer the case for a partial interpretation. The partial interpretation of a function $f/n$ is as a mapping from $D^{n+1}$ to $\{\mathbf{t},\mathbf{f},\mathbf{u}\}$. As for predicates, we can distinguish true tuples $f^I_{ct}$, false tuples $f^I_{cf}$, and unknown tuples $f^I_u$. While a functional dependency holds in the set $f^I_{ct}$, this is not the case for the latter two sets. However, $(\overline{d},d) \in f^I_{ct}$ iff $(\overline{d},d') \in f^I_{cf}$ for all $d' \in D \setminus \{d\}$.

If $I$ is a partial structure, $U$ a set of domain atoms, and $v$ a truth value, we use $I[U:v]$ to refer to the partial structure that equals $I$ except that for each domain atom $P(\overline{d}) \in U$, it holds that $P(\overline{d})^{I[U:v]} = P^{I[U:v]} = v$.

The partial structure $I$ that corresponds to a two-valued v $I$ is such that, for predicate symbols $P/n$, $P^I_{ct} = P^I$, $P^I_{cf} = D^n \setminus P^I_{ct}$ and, $P^I_u = \emptyset$ and, for function symbols $f/n$, $f^I_{ct} = \{(\overline{d},d) \mid f^I(\overline{d}) = d\}$, $f^I_{cf} = D^{n+1} \setminus f^I_{ct}$ and $f^I_u = \emptyset$.

The *truth order* $<_t$ on truth values is induced by $\mathbf{f} <_t \mathbf{u} <_t \mathbf{t}$. The *precision order* $<_p$ on truth values is induced by $\mathbf{u} <_p \mathbf{t}$ and $\mathbf{u} <_p \mathbf{f}$. This order is extended to a precision order over partial structure. A partial structure $I$ is less precise than a partial structure $I'$ (notation $I \leq_p I'$) if, for all symbols $s \in \Sigma$, $s^I_{ct} \subseteq s^{I'}_{ct}$ and $s^I_{cf} \subseteq s^{I'}_{cf}$. Maximally precise partial structures are two-valued.

In the remainder of the chapter, partial interpretation or structure is intended when interpretation or structure is used.

### 1.2.2 Partial Functions

In standard logic, function symbols denote total functions. In practice, partial functions are unavoidable, e.g., a function that maps persons to their spouse is naturally undefined for singles as well as for objects that are not a person, and the arithmetic operation division is undefined when the denominator is zero. So, our logic supports partial functions; however, defining a semantics for partial functions gives rise to undefined terms (also called non-denoting); this is a subject of controversy [Frisch and Stuckey 2009, Wittocx 2010].

The simplest solution is to restrict the syntax of formulas. One could, e.g., only allow terms of the form $f(t)$ in contexts where it is certain that $f(t)$ is defined. This option is often taken in mathematics, where terms like, e.g., $1/0$ are considered meaningless, but quantifications of the form $\forall x : x \neq 0 \Rightarrow 1/x \neq 42$ are allowed as it is clear that the division $1/x$ will be defined for all relevant $x$. This idea has been implemented for example in the Rodin toolset for Event-B [Abrial et al. 2010], where for every occurrence of a partial function, it should be provable that the function will only by applied to values in its domain. However, this approach is too restrictive for a KBS. For example, in planning problems, the function $Do/1$ in a term $Do(t)$ that refers to the action performed at time $t$ is typically a partial one. It can be pretty hard to come up with the right $Condition/1$ predicate such that one can write $\forall t : Condition(t) \Rightarrow \ldots Do(t) \ldots$. It is entirely impossible, when the partial function is a constant for which it is a priori unclear whether it is defined, such as *Unicorn*. So we allow terms in contexts where they can be undefined. This brings us to the question what an ambiguous form such as $White(Unicorn)$ means. Does it mean "if the unicorn exists then it is white", i.e., $\forall x : Unicorn = x \Rightarrow White(x)$ or "the unicorn exists and is white", i.e., $\exists x : Unicorn = x \wedge White(x)$ (which equals $(\exists x : Unicorn = x) \wedge White(Unicorn)$). For the user, having to write the longer unambiguous form is rather inconvenient (especially in case of nested partial functions).

The current approach, which is the result of some years of experimenting, is based on the *relational semantics* proposed in [Frisch and Stuckey 2009]. It turns out to be flexible, intuitive and to allow for elegant modeling. The ambiguous form $A(f(\bar{t}))$ is given the second of the above two meanings, namely $\exists x : f(\bar{t}) = x \wedge A[x]$ or equivalently $\exists x : f(\bar{t}) = x \wedge A[f(\bar{t})]$. When the user is in doubt or prefers the other form, he should avoid the ambiguous form and explicitly write one of the explicit forms.

In a two-valued structure $I$, the value of a partial function $f/n$ is a mapping $f^I : S \to D$ where $S$ is some subset of $D^n$. In a partial structure $I$, $f/n$ is undefined for $\overline{d}$ if and only if $(\overline{d}, d) \in f^I_{cf}$ for all $d \in D$. We say that the image $f(\overline{d})$ is *undefined* when $f$ is undefined for $\overline{d}$. The interpretation of a term with a direct subterm that is undefined is also undefined; that of an atom with a direct subterm that is undefined is *false*. This corresponds to the above described semantics.

### 1.2.3 Arithmetic

Standard FO can easily be extended with arithmetic. Indeed, numbers can be added to the domain and various (partial) functions can be included in the vocabulary to perform arithmetic.

So far, the IDP system only supports arithmetic over integers. This is our motivation to include the integers in every domain of FO(ID,AGG,PF,T) and the arithmetic partial functions +/2, -/2, -/1, */2 (multiplication), //2 (division) %/2 (modulo) and $abs/1$ in every

vocabulary[4]. To refer to these integers, every vocabulary also includes the constant symbol $n$ for every integer $n$. Furthermore, in every structure, the interpretation of these integer constants is fixed to the corresponding integer in the domain.

### 1.2.4 Aggregates

Aggregates are an important language construct to boost the expressiveness of first-order logic. FO(ID,AGG,PF,T) includes the aggregate functions *cardinality*, *sum*, *product*, *minimum* and *maximum*. The basic underlying concept is the set expression $\{(\bar{x}) \mid \varphi\}$ or $\{(\bar{x},t) \mid \varphi\}$ where $\bar{x}$ is a tuple of new variables and $t$ is a term $t[\bar{x},\bar{y}]$ with variables in $\bar{x}$ and in the free variables $\bar{y}$ of the expression. Given a two-valued structure $I$, such a set expression denotes the set of tuples $\{(\bar{x})^I \mid \varphi^I = \mathbf{t}\}$ or $\{(\bar{x},t)^I \mid (\exists z : z = t \wedge \varphi)^I = \mathbf{t}\}$, where the existential quantification ranges over the domain of $t$. The inclusion of $\exists z : z = t$ in the set prescription disambiguates the set in case of partial functions in $\varphi$.

Cardinality expressions are written in FO(ID,AGG,PF,T) as $\#\{\bar{x} : \varphi\}$ and this term denotes the number of tuples in the set. For the other four, the aggregate expressions take the form $sum\{(\bar{x},t) : \varphi\}$, $prod\{(\bar{x},t) : \varphi\}$, $min\{(\bar{x},t) : \varphi\}$, and $max\{(\bar{x},t) : \varphi\}$ respectively. These expressions denote the value of the aggregate function on the multiset obtained by extracting the last element $t^I$ of each tuple. For instance $sum\{((x_1,x_2),x_2) : P(x_1,x_2)\}$ sums the values of the second element of all tuples in $P$. If there are multiple occurrences of the same "second element" in tuples of $P$, then they are counted according to their multiplicity.

All aggregate functions are partial functions; they are undefined for infinite sets. Moreover, all aggregate functions except # are only defined for sets containing a tuple $(\ldots,d)$ with $d$ an integer. The aggregates *min* and *max* are also undefined for the empty set; in contrast, *sum* and *prod* map the empty set to 0 and 1 respectively.

The aggregates supported by FO(ID,AGG,PF,T) also include the class of formulas $\exists_{\sim n} x : \varphi$ with $n$ a natural number and $\sim$ one of the comparison operators. While such formulas are equivalent with $\#\{x : \varphi\} \sim n$, they are more concise and a convenient extension of existensial quantification. They allow one to express "there exists exactly $n$" $(= n)$, "at most $n$" $(\leq n)$, "less than $n$" $(< n)$, "more than $n$" $(> n)$, "at least $n$" $(\geq n)$, and "there does not exist exactly $n$" $(\neq n)$ values for $x$ such that $\varphi$ holds. Note that $\exists_{\geq 1} x : \varphi$ is equivalent with $\exists x : \varphi$, and $\exists_{=0} x : \varphi$ with $\neg(\exists x : \varphi)$.

### 1.2.5 Definitions

The logic FO(ID,AGG,PF,T) contains a definition construct to express different kinds of (possibly inductive) definitions. This construct is one of the most original aspects of FO(ID, AGG,PF,T) and we explain it in more detail. For additional details we refer to [Denecker and Ternovska 2008].

[4] The current implementation of the IDP language, described in Section 1.4, has only limited support for integer division.

Definitions are important building blocks of any scientific theory. However, there is no general way to express inductive/recursive definitions in FO. Though the notion of definition is informal, definitions have some extraordinary properties. Certainly those used in formal mathematical text strike us for the precision of their meaning. The formal semantics of FO(ID,AGG,PF,T) definitions carefully formalizes this meaning. Several types of informal definitions can be distinguished. Below, the three most common ones are illustrated: Example 1.2.1 is a non-recursive one, Example 1.2.2 is a monotone one, while Example 1.2.3 is by induction over a well-founded order, namely over the subformula order. Definitions over a well-founded order frequently contain non-monotone rules. For instance the rule defining $I \models \neg\alpha$ has a non-monotone condition $I \not\models \alpha$.

**Example 1.2.1.** Let $a, b$ and $c$ be integers; $a$ is between $b$ and $c$ iff $b \leq a$ and $a \leq c$.

**Example 1.2.2.** Let $(N,E)$ be a graph with nodes $N$ and edges $E$. The transitive closure $T$ of $(N,E)$ is defined inductively as follows.

- If $(a,b) \in E$, then $(a,b) \in T$,
- If for some $c \in N$, it holds that $(a,c) \in T$ and $(c,b) \in T$, then also $(a,b) \in T$.

**Example 1.2.3.** Let $I$ be a two-valued structure of a propositional vocabulary. The satisfaction relation $\models$ is defined by induction over the structure of formulas:

- $I \models P$ if $P \in I$.
- $I \models \alpha \wedge \beta$ if $I \models \alpha$ and $I \models \beta$.
- $I \models \alpha \vee \beta$ if $I \models \alpha$ or $I \models \beta$ (or both).
- $I \models \neg\alpha$ if $I \not\models \alpha$.

In FO(ID,AGG,PF,T), a *formal* definition $\Delta$ is a set of rules of the form $\forall \bar{x} : P(\bar{t}) \leftarrow \varphi$ or $\forall \bar{x} : f(\bar{t}) = t' \leftarrow \varphi$, with the free variables of $\varphi$ and the variables in $\bar{t}$ and $t'$ amongst the $\bar{x}$. We refer to $P(\bar{t})$ and $f(\bar{t}) = t'$ as the *head* of the rule and to $\varphi$ as the *body*. In the first form, $P$ is the defined symbol; in the second, $f$ is. The defined symbols of $\Delta$ are all symbols that are defined by at least one of its rules; all other symbols occurring in $\Delta$ are called *parameters* or *open* symbols of $\Delta$. Intuitively, for each two-valued structure of the parameters, $\Delta$ determines the interpretation of the defined symbols in a unique way. For instance, the definition of transitive closure can be formalized as follows

$$\left\{ \begin{array}{l} \forall a,b : T(a,b) \leftarrow E(a,b). \\ \forall a,b : T(a,b) \leftarrow \exists c : T(a,c) \wedge T(c,b). \end{array} \right\}$$

The different sorts of definitions have different semantic properties. It is commonly assumed that the defined set is the *least* set that satisfies the rules of the definition, i.e., the least set such that the head is true whenever the body is true. However, this is only true for monotone definitions. It does not hold for non-monotone definitions as the following example, from

[Denecker and Vennekens 2014], illustrates.

$$\left\{ \begin{array}{l} Even(0). \\ \forall x : Even(x+1) \leftarrow \neg Even(x). \end{array} \right\}$$

Intuitively, this defines the infinite set $\{Even(0), Even(2), Even(4) \ldots\}$ of even numbers. However, also the infinite set $\{Even(0), Even(2), Even(3), Even(5), \ldots\}$ satisfies the rules as for every rule instance with a true body, the head is also true. Both solutions are minimal; however, none is "least".

Still, there is an explanation that applies to all kind of definitions [Buchholz et al. 1981]: the set defined by an inductive definition is the result of a construction process. The construction starts with the empty set, and proceeds by iteratively applying non-satisfied rules, till the set is saturated. In the case of monotone definitions, rules can be applied in any order; but in the case of definitions over a well-founded order, rule application must follow the well-founded order. This condition is necessary for the non-monotone rules. If they would be applied too early, later rule applications may invalidate their condition. E.g., in the initial step of the construction of $\models$, when the relation is still empty, we could derive $I \models \neg\varphi$ for each $\varphi$, but the condition $I \not\models \varphi$ will in many cases later become invalidated. The role of the induction order is exactly to prevent such an untimely rule application. E.g., to prevent deriving $Even(3)$ before $Even(2)$ has been derived.

The problem we face in formalizing this idea for the semantics of FO(ID,AGG,PF,T) definitions, is that the syntax of FO(ID,AGG,PF,T) does not specify an explicit induction order for non-monotone FO(ID,AGG,PF,T) definitions. Thus, the question is whether one can somehow "guess" the induction order. Indeed, if we look back at the definition of Example 1.2.3, we see that the order is implicit in the structure of the rules: formulas in the head of rules are always larger in the induction order than those in the body. This holds true in general. It should be possible then to design a mathematical procedure that somehow is capable to exploit this implicit structure.

In [Denecker and Vennekens 2014], this idea was elaborated. The induction process of an FO(ID,AGG,PF,T) definition is formalized as a sequence of three-valued structures of increasing precision. Such a structure records what elements have been derived to be in the set, what elements have been derived to be out of the set, and which have not been derived yet. Using the current three-valued structure, one can then establish whether it is safe to apply a rule or not. All induction sequences can be proven to converge. In case the definition has the form of a logic program and the underlying structure is a Herbrand interpretation, the resulting process can be proven to converge to the well-known well-founded model of the program [Van Gelder et al. 1991]. As such, the semantics of FO(ID,AGG,PF,T) definitions is a generalization of the well-founded semantics, to arbitrary bodies, arbitrary structures and with parameters. This (extended) well-founded semantics provides a uniform formalization for the two most common forms of induction (monotone and over a well-founded order) and

even for the less common form of iterated induction [Buchholz et al. 1981]. Compared to other logics of iterated inductive definitions, e.g., the work in [Buchholz et al. 1981], the contribution is that the order does not have to be expressed; a substantial advantage as this can be very tedious.

The satisfaction relation of FO is thus extended to handle definitions by means of the well-founded semantics [Van Gelder 1993], since it formalizes the informal semantics of rule sets as inductive definitions [Denecker 1998, Denecker and Vennekens 2014, Denecker et al. 2001]. We now formally describe how this is done. First, consider definitions $\Delta$ that define only predicate symbols. We use the parametrised well-founded semantics. This semantics has been implicitly present in the literature for a long time, by assigning a meaning to an *intensional* database. We follow the formalisation by Denecker and Vennekens [Denecker and Vennekens 2007]. We say that a two-valued structure $I$ satisfies $\Delta$ ($I \models \Delta$) if $I$ is the parametrised well-founded model of $\Delta$, that means that $I$ is the well-founded model of $\Delta$ when the open symbols/parameters are interpreted as in $I$.

Checking the latter is done by computing the well-founded model of $\Delta$. This can be computed as the limit of a *well-founded induction* [Denecker and Vennekens 2007], defined below.

**Definition 1.2.4** (Refinement)**.** We call a partial structure $I'$ a *refinement* of partial structure $I$ if one of the following holds:

- for every defined predicate $P$ and every tuple of domain elements $\overline{d}$,

$$P(\overline{d})^{I'} = \max_{\leq_t}\{\varphi[\overline{x}/\overline{d}]^I \mid \forall \overline{x} : P(\overline{x}) \leftarrow \varphi \text{ is a rule in } \Delta\}.$$

- $I' = I[U : \mathbf{f}]$, where $U$ is a set of domain atoms unknown in $I$ such that for every $P(\overline{d}) \in U$ and every rule $\forall \overline{x} : P(\overline{x}) \leftarrow \varphi$ in $\Delta$, it holds that $\varphi[\overline{x}/\overline{d}]^{I'} = \mathbf{f}$ (such a set $U$ is called an unfounded set of $\Delta$ in $I$ [Van Gelder et al. 1991]).

A refinement is *strict* if $I' \neq I$.

The first refinement evaluates all rule bodies of all defined domain atoms and assigns the largest truth value (e.g., if one rule derives an atom to be true, and the second rule does not yet derive information about that atom (**u**), the atom obtains the value **t**) to each defined atom. The second refinement identifies an unfounded set: a set of domain atoms such that the bodies of rules defining them can only become true if at least one of these atoms is true in the first place (due to cyclic dependencies). Such atoms can never be derived constructively using the definition, hence they must be false.

**Definition 1.2.5** (Well-founded induction)**.** Let $I$ be a partial structure that interprets only the open symbols of $\Delta$. A *well-founded induction* of $\Delta$ in $I$ is a sequence $(I_i)_{i\leq n}$, with $n \in \mathbb{N}$, of partial structures such that the following hold:[5]

- $I_0 = I$;
- for each $i < n$, $I_{i+1}$ is a refinement of $I_i$.

A well-founded induction is *terminal* if its limit $(I_n)$ has no strict refinements.

Denecker and Vennekens [Denecker and Vennekens 2007] showed that all terminal well-founded inductions in $I$ have the same limit, namely the well-founded model of $\Delta$ in context $I$.

To extend this satisfaction check to definitions defining functions, one treats a function $f/n$ as if it is defining an $n+1$-ary relation. For what concerns the use of partial functions in the head of a rule, note that $\forall \bar{y} : p(\bar{t}) \leftarrow \varphi$ is equivalent with $\forall \bar{x}\bar{y} : p(\bar{x}) \leftarrow \bar{x} = \bar{t} \wedge \varphi$. Partial functions in bodies have the same meaning as in formulas.

In FO(ID,AGG,PF,T), a definition is seen in a pure declarative way, as a proposition stating a special logical relationship between defined predicates and parameter symbols. In case a theory contains multiple definitions of the same predicate, the theory states multiple independent such propositions. For instance, a theory that contains

$$\left\{ \begin{array}{l} \forall x : Human(x) \leftarrow Man(x) \vee Woman(x). \end{array} \right\}$$
$$\left\{ \begin{array}{l} \forall x : Human(x) \leftarrow Child(x) \vee Adult(x). \end{array} \right\}$$

states that Human is the union of men and women and also that human is the union of children and adults. This implies for example that the union of men and women, and of children and adults is identical. Note that this declarative view implies that the definition

$$\left\{ \begin{array}{l} p \leftarrow q. \\ q \leftarrow p. \end{array} \right\}$$

has a different meaning than the pair of definitions

$$\left\{ \begin{array}{l} p \leftarrow q. \end{array} \right\}$$
$$\left\{ \begin{array}{l} q \leftarrow p. \end{array} \right\}$$

Indeed, in the former, $p$ and $q$ are false in the only well-founded model. In the latter, the structure in which $p$ and $q$ are true is also a model since we now have two definitions, each with a parameter.

To reason with definitions, the IDP solver makes also use of their completion. The *completion* of $\Delta$ for a symbol $P$, defined in $\Delta$ by the rules $\forall \bar{x}_i : P(\bar{t}_i) \leftarrow \varphi_i$ with $i \in [1,n]$,

[5] In the infinite case, a similar sequence can be constructed. For details, see [Denecker and Vennekens 2007].

is the set consisting of the sentence $\forall \bar{x}_i : \varphi_i \Rightarrow P(\bar{t}_i)$ for each $i \in [1,n]$ and the sentence $\forall \bar{x} : P(\bar{x}) \Rightarrow \bigvee_{i\in[1,n]}(\bar{x} = \bar{t}_i \wedge \varphi_i)$; the completion for defined function symbols is defined similarly. This set is denoted as $comp_{P,\Delta}$, the union of all these sets for $\Delta$ as $comp_\Delta$. It is well known (see, e.g., [Van Gelder et al. 1991]) that if $I \models \Delta$ then $I \models \text{comp}_\Delta$ but not always vice-versa (e.g., the inductive definition expressing transitive closure is stronger than its completion).

### 1.2.6 Types

While first-order logic is untyped, the real world is typed. For example, in a course scheduling problem, we can distinguish persons, which can be further divided in teachers and students, courses, rooms, time slots, etc. The advantages are well known in the field of programming languages. For example, the introduction of the book of Pierce [Pierce 2002] lists, among others, early detection of errors and a minimum of documentation. These advantages also hold for a knowledge representation language. Moreover, the use of types leads to more detailed and more accurate modeling of the different types of objects in the domain of discourse.

FO(ID,AGG,PF,T) is a simple *order sorted logic*. This means that a vocabulary contains a set of types on which a subtype relation is defined. The corresponding type hierarchy is a set of trees. Two types are *disjoint* if they have no common supertype (a type is a supertype of itself). Every vocabulary includes the type *int* of the integers and its subtype *nat* of the natural numbers. All predicate and function symbols of a vocabulary are typed by means of a type signature which associates a type with each argument position, and, in the case of functions, with the result. All variable occurrences in a theory are typed; all occurrences of a variable within the same scope have the same type; the type of a variable is given when it is introduced in a formula or set expression, for example, $\forall x[T] : \varphi[x]$ and $\#\{x[T]\ y[T'] : \varphi[x,y]\} < 3$.

In a structure, a domain $D_T$ is associated with each type $T$; for a subtype $T_1$ of $T_2$, $D_{T_1}$ is a subset of $D_{T_2}$; if $T_1$ and $T_2$ are disjoint types, $D_{T_1}$ and $D_{T_2}$ must be disjoint. Structures are well-typed. This means that for a predicate symbol $P$ of type $(T_1,\ldots,T_n)$, its value $P^I$ belongs to the Cartesian product $D_{T_1} \times \cdots \times D_{T_n}$, and for a function symbol $f$ from $(T_1,\ldots,T_n)$ to $T$, its value $f^I$ is a partial function from $D_{T_1} \times \cdots \times D_{T_n}$ to $D_T$. For the evaluation of quantified formulas $\forall x[T] : \varphi$ and $\exists x[T] : \varphi$, enumeration of the values of $x$ is over the domain $D_T$. Similarly, in a set expression $\{(\bar{x}[\bar{T}],t)|\varphi\}$ of an aggregate, each $x_i$ is assigned domain elements from $D_{T_i}$. Note that a term $t$ of type $T_1$ that occurs in an argument of an atom or function where a term of type $T_2$ is expected can have the meaning of an undefined term (Section 1.2.2). Indeed, the atom evaluates to false or the function is undefined when $t$ evaluates to a domain element outside $D_{T_2}$. While this cannot happen when $T_2$ is a supertype of $T_1$, it always is the case when $T_1$ and $T_2$ are disjoint; it depends on the evaluation when there is a third type that is a supertype of both. When the types are disjoint, it is appropriate to raise a type error as this is likely a design error in the logical theory.

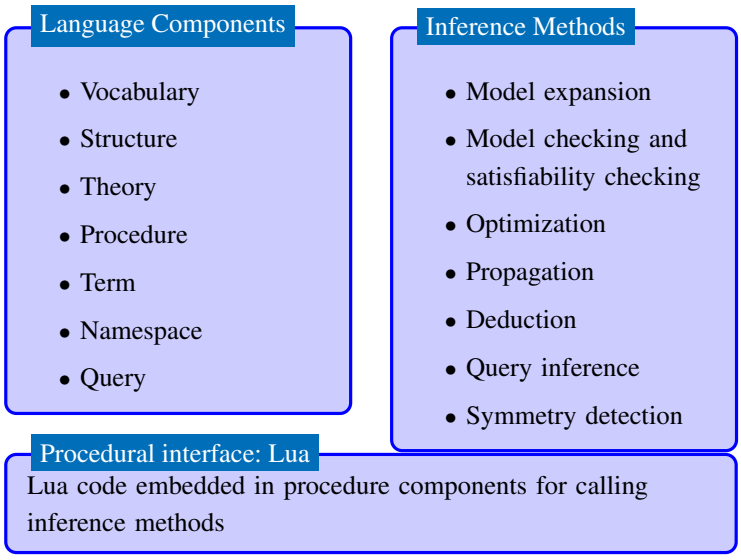


**Figure 1.1** High-level representation of a KBS system

## 1.3 IDP as a Knowledge Base System

We start the section with a description of the architecture and a discussion of design decisions. We finish with sketching an application where the same knowledge is used for different tasks.

### 1.3.1 Architecture and Design Decisions

Here, we introduce and motivate the basic design decisions underlying the IDP system, the decisions that determine the look and feel of IDP as a KBS. IDP is an implementation of a KBS. Besides the two main parts, the *declarative language* and the *inferences methods*, there is also a part that provides *procedural integration*. An overview is shown in Figure 1.1.

The first design decision, the one most visible to users, is about the language of the KBS. The language should be (i) *rich* enough so that users can express all their needs; (ii) *natural* enough so that theories stay close to the original (natural language) problem statement and are easy to read and to debug; and (iii) *modular* enough to allow for reuse and future extensions.

It is sometimes argued that the expressiveness of a language should be limited, to avoid that the language becomes undecidable or intractable. We disagree. First, note that decidability and tractability depend on the task at hand. While deduction in first order logic is undecidable, other forms of inference, such as model expansion and querying in the context of a finite domain, are decidable. Second, while a more expressive language might allow users to express tasks high in the polynomial hierarchy, that does not imply that simple tasks become harder to solve. Rather to the contrary, stating the problem in a richer language sometimes allow the

KBS to exploit structural information that would be hidden in a more lower-level problem statement.

To address the requirement of a rich and a natural language, we have opted for FO(ID, AGG,PF,T), FO extended with definitions, aggregates, partial functions and types. We choose for first-order logic because conjunction, disjunction, universal and existential quantification have very natural meanings. Extensions are needed because FO has various weaknesses. Inductive definitions overcome the weakness that FO cannot express inductively defined concepts. Also non-inductive definitions are very useful. In mathematical texts, it is common practice to use "if" when defining concepts; this "if" here is (in natural language) a *definitional implication*. Aggregates allow users to concisely express information requiring lengthy and complex FO formulas. Types are omnipresent in the context of natural language, where quantification typically refers to a specific set of objects (e.g., everyone is mortal). For integration with a procedural language, IDP currently offers an interface to the languages Lua and C++.

The third language requirement, modularity, is important both at the language and at the system level. An advantage of first-order logic as basis of our language is that language extensions can be added without much interference at the syntactical and semantical level. For example to introduce aggregates to FO, it suffices to extend the satisfaction of atoms in which an aggregate occurs in order to obtain a semantics for a language extended with aggregates.

On the system level, we have also attempted to organize inference engines in a modular way so that components can be reused in multiple engines. For example, the model expansion inference is currently implemented as ground-and-solve; the solver can be used separately from the grounder, and the grounding phase is composed of several smaller, reusable parts (for example, evaluation of input-*-definitions [Jansen et al. 2013]). Also various approaches to preprocess simple theories in order to improve their computational efficiently are integrated in the system. Examples are symmetry detection [Devriendt et al. 2016b] and symmetry breaking [Devriendt et al. 2012] methods and the use of deduction to detect functional dependencies [De Cat and Bruynooghe 2013]. Such preprocessing techniques also improve the user-friendliness and robustness of the KBS as a whole. Indeed, they let a user focus on the declarative modeling, and partly relieve the user from the task of fine-tuning it on a specific solver.

### 1.3.2 Multiple Inference Methods Within One Application Domain

Given any knowledge base, there are often multiple applications that require different kinds of inference. By way of example, we explore the setting of a university course management system. Its input is a database with information on students, professors, classrooms, etc. One task of the system is to help students choose their courses satisfying certain restrictions. Such an application is usually interactive; students make choices and, in between, the system checks

the knowledge base. It removes choices when they become invalid, adds required prerequisites when a course is selected, etc; this is an example of *propagation* inference. When the student has made all choices he deems important, the system could use *the same* knowledge base to complete the student's choices to obtain a complete schedule. This type of inference is called *model generation* or *model expansion*: one starts with partial information (certain selections have been made) and wants to extend it into a complete solution, namely a model of the course selection theory. Another task where model expansion is needed in the same application area is to generate a schedule where every course is assigned a location and a starting time such that 1) no person has to be at two places at the same time, 2) no room is double-booked and 3) availability of professors is taken into account. However, due to the large number of optional courses, such a solution (in which no student has overlapping courses) will probably not exist. In this case, we might want to find a solution in which the number of conflicts is minimal; this requires *minimization* inference. Now, one might want to mail students with schedules with overlaps to give them the opportunity to change their selection. Hence, the solution of the minimization inference should be *queried* to find the overlapping courses for every student. In the course of a semester, professors might have to cancel a lecture due to other urgent obligations. In that case, we want to find a *revision* of the current schedule, taking the changed restrictions into account and minimizing the number of changes with respect to the current schedule. In case such revisions are done manually, the *model checking* inference can be used to ensure that no new conflicts are introduced. If some conflict does occur, an *explanation* should be provided. Finally, if a valid schedule is found, a *visualization* inference can be used to create an easy-to-understand, visual representation of the schedule, personalized by the viewer's status (student, professor, administrative personnel, etc.). Part of such an application, using the IDP system as a back end, is shown at http://krr.bitbucket.org/courses.

## 1.4 The IDP Language

The IDP language is the input language of the IDP system. A program in the IDP language consists of declarative and imperative components. The declarative components are *vocabulary*, *structure*, *theory* and *term* components. Together, they provide a concrete computer-readable syntax for FO(ID,AGG,PF,T). The imperative components allow one to perform computational tasks. They consists of *procedures*. Each procedure embeds a piece of imperative Lua [Ierusalimschy et al. 1996] code; besides performing standard imperative operations, procedures can apply inference methods upon FO(ID,AGG,PF,T) theories encoded in the declarative components.

Vocabulary, structure and theory components are described in Section 1.4.1; procedure components in Section 1.4.2 and term components in Section 1.4.3. In this section we do not strive for completeness but focus on what is needed to get started using the IDP system and on providing answers to the difficulties a starting user might have.

Before describing the different kinds of components, we first discuss a few general notational conventions. Names are everywhere, they are used for types, predicates, functions (including constants), and variables as well as for domain elements in structures. To distinguish them from numbers, they start with a Latin letter (upper or lower case) and consist of a sequence of Latin letters and digits; also a few special characters such as "_" are allowed. For domain elements, one can deviate from this convention by using a string notation. For details, we refer to the manual [Bogaerts et al. 2012]. Comments that fit on a single line start with "//". One can start longer comments with "/*" and end them with "*/".

### 1.4.1 The Logic

#### 1.4.1.1 Vocabulary

The vocabulary of an FO(ID,AGG,PF,T) theory is represented as a vocabulary component. We start with an example.

```
vocabulary courses {
  type course
  type person
  type student isa person
  type instructor isa person
  type age isa nat
  takes(student, course)
  hasAge(person):age
  Vacation
  Boss: person
}
```

A vocabulary declaration takes the form "**vocabulary** ⟨vocname⟩ { ⟨typed symbol list⟩}". It specifies the name of the vocabulary, here courses and its symbols. The symbol list comprises type symbols, and typed predicate and function symbols. Each symbol is to be declared on a new line. Types are declared using the keyword "**type**". A type may be declared as a subtype using the keyword "**isa**" followed by a comma-separated list of supertypes. For example "**type** A **isa** B, C" declares *A* as a direct subtype of *B* and *C*. The declared **isa** graph needs to be acyclic. The integers, type **int**, and its subtype, the natural numbers, type **nat**, are part of every vocabulary and need not be declared. The same holds for predicates and functions that are part of every FO(ID,AGG,PF,T) theory such as comparison predicates and arithmetic functions. Predicates and functions are introduced by declaring their signature. In the example, takes is a relation over student and course and hasAge is a function from person to the subtype age, a subtype of nat. The symbol Vacation is a propositional symbol, and Boss a constant symbol. Partial functions are introduced by the keyword **partial**, for example **partial** hasAge(person) :age would declare hasAge as a partial function. The IDP system cannot yet cope well with infinite types, so **int** and **nat** can better be avoided in signatures of predicates and functions.

#### 1.4.1.2 Structure

A structure component describes a partial structure for a vocabulary, in particular the domains of the user declared types. We start again with an example.

```
structure data1: courses{
  course = {Logic; Math}
  student = {John; Bob; Alice}
  instructor = {Marc; Gerda; Maurice}
  person = {John; Bob; Alice; Marc; Gerda; Maurice}
  age = {1 .. 65}
  takes<ct> = {John,Logic}
  takes<cf> = {Bob,Math}
  hasAge<ct>= {John->25; Bob -> 30; Alice->19}
  Vacation=true
  Boss=Alice
}
```

A structure has a name, here data1, and specifies the vocabulary that it (partially) interprets, here courses. It specifies an assignment of values for symbols using a list of "symbol=value" equations. Boolean values are denoted true or false, as illustrated by the equation Vacation= true in the example. Other base values are numbers, strings, or user-defined domain values like Bob, Math. Set values are denoted as "{ ⟨semicolon separated list⟩ }". The list may consist of individual entities or of tuples of individual entities. Tuples are denoted as comma separated lists of domain values, potentially between parentheses "( ...)". The shorthand "n..m" enumerates the integer interval $[n,m]$, as illustrated for age. The same holds for characters, e.g., student = {a..d} is a shorthand for student = {a;b;c;d}.

A structure specifies, implicitly or explicitly, the value of each type of the vocabulary[6]. A value of a type is a set of domain elements. User-defined domain elements are identified by symbolic identifiers (e.g., Bob, Math) but these identifiers are not symbols of the vocabulary and cannot be used in the theory. One can use integers as names for the domain elements of types, even if this type is not a subtype of **int**. For example student = {1..50} introduces 50 students. Note, these domain elements are not integers, the domain element 49 of type student is different from domain element 49 of type integer.

A (total) value for a predicate symbol is a set of domain elements or tuples of it. Alternatively, a structure may specify a partial value for a predicate symbol P, as an assignment of a list of certainly true, certainly false and unknown tuples to respectively P<ct>, P<cf>, and P<u>. Only two out of three need to be specified. This is illustrated by takes.

The value or partial value for functions is specified in an analogous way, with the difference that the user is allowed (but not obliged) to specify a tuple "(a1,...,an,b)" of a function in the

[6] Autocompletion may derive missing type domains; e.g., in the absence of a domain for age, autocompletion will derive {19, 25, 30} for it, in absence of a domain for person, it will derive the union of the student and instructor domains.

form "a1 ,.., an −> b". For partial interpretations of functions, we refer to the discussion on partial structures in Section 1.2.1. For example, hasAge<cf> = {Marc −> 1; Marc −> 2} can be used to express that the age of Marc is neither 1 nor 2.

In IDP one cannot currently use a constant as a bound in a domain enumeration, as in student = {1.. nbstudents}, where nbstudents is a constant symbol whose value is specified elsewhere. Another limitation that was already mentioned earlier, is that domain values such as Bob and Alice (identifiers introduced at the right side of "symbol=value" equations), are not part of the vocabulary and cannot appear in theories.

Recall that there exist also many interpreted symbols (e.g., numerical operators and aggregates) whose values are fixed and are implicit parts of all structures.

#### 1.4.1.3 Theory

A theory component over some vocabulary is declared as "**theory** ⟨theory name⟩ : ⟨vocname⟩ { ... }". For the syntax of formulas and definitions, that of the formal base language is followed as closely as possible. Formulas and rules are terminated with a "." and rules are grouped in definitions which are put between "**define** {" and "}". The following table provides a translation in a more keyboard friendly notation[7].

| FO(ID,AGG,PF,T) | IDP language | FO(ID,AGG,PF,T) | IDP language |
|---|---|---|---|
| $\wedge$ | & | $\geq$ | >= |
| $\vee$ | \| | $=$ | = |
| $\Rightarrow$ | => | $\neq$ | ~= |
| $\Leftarrow$ | <= | $\leftarrow$ | <- |
| $\Leftrightarrow$ | <=> | $\#\{\bar{x} : \varphi\}$ | #{x1 ... xn : φ} |
| $\neg$ | ~ | $sum\{(\bar{x},t) : \varphi\}$ | sum{x1 ... xn : φ: t} |
| $\forall$ | ! | $prod\{(\bar{x},t) : \varphi\}$ | prod{x1 ... xn : φ: t} |
| $\exists$ | ? | $max\{(\bar{x},t) : \varphi\}$ | max{x1 ... xn : φ: t} |
| $\leq$ | =< | $min\{(\bar{x},t) : \varphi\}$ | min{x1 ... xn : φ: t} |

By way of example, we show a vocabulary, a structure and a theory for a small graph problem that formalizes a connected graph over a set of nodes.

```
vocabulary V{
  type Node
  Forbidden(Node,Node)
  Edge(Node,Node)
  Reachable(Node)
  Root:Node
}
```

[7] The IDE at http://dtai.cs.kuleuven.be/krr/idp-ide/ visualizes the symbols in the syntax of FO(ID,AGG,PF,T).

```
structure S:V{
  Node = {A..D}
  Forbidden = {A,A; A,B; A,C; B,A; B,B; B,C; C,C; C,D; D,D}
  Root = A
}

theory T: V{
  // inductive definition of Reachable
  define {
    Reachable(Root).
    !x [Node]: Reachable(x) <-
               ?y [Node]: Reachable(y) & Edge(y,x).
  }

  // The graph is fully connected
  !x [Node] : Reachable(x).

  // No forbidden edges
  !x[Node] y[Node] : Edge(x,y) => ~Forbidden(x,y).
}
```

The theory contains a definition[8] of the Reachable predicate and two formulas that constrain the solution. All variables are typed with type Node; however, these types can be omitted since *type inference* will derive them from the signatures of the symbols in which the variables occur.[9] Observe that this theory defines Reachable as the transitive closure of parameter Edge. This definition is syntactically similar to a Prolog program, but unlike a Prolog program, it is here the defined predicate that is known (it is the set of all nodes) and the parameter predicate that is unknown. It illustrates the declarative understanding of a definition that expresses a particular logical relationship between the parameters and the defined symbols, and not a way to compute the defined symbols in terms of the parameters.

### 1.4.2 Procedure

A procedure component is a chunk of Lua code [Ierusalimschy et al. 1996] encapsulated in the form of an IDP component (a keyword **procedure**, a name, a list of parameters and the chunk of code between "{" and "}"). When the IDP system is run, it calls the procedure main( ). Typically, one will use the IDP system to do some reasoning on an FO(ID,AGG,PF,T) theory. Here is a simple example:

[8] The **define** keyword is optional; it emphasizes that the brackets { and } are delimiters of a definition.

[9] If within the same scope a variable appears in argument positions with different types, the inferred type is their least supertype if it exists, otherwise a type error is raised, as explained in Section 1.2.6. If no type can be inferred, e.g., as in !x: x=x, also then an error is raised.

```
procedure main(){
    stdoptions.nbmodels = 0
    printmodels(modelexpand(T,S))
}
```

The first line, stdoptions.nbmodels = 0, configures the IDP system to compute all models (with a positive number $n$, inference is stopped after $n$ models have been found; default value is 1). The second line calls IDP's modelexpand procedure with as input arguments the theory $T$ and the structure $S$ and prints these models (printmodels). The modelexpand procedure calls upon the solver of the IDP system to search for models of $T$ that expand the input structure $S$ and returns an array of models. To print a single model, one can select a model and print it, e.g. print(modelexpand(T,S)[3]) to print the third model. With the theory and structure as given in the small graph theory of the previous section, the above procedure prints 24 models, the first one being:

```
structure : V {
  Node = { "A"; "B"; "C"; "D" }
  Edge = { "A","D"; "B","D"; "C","A"; "C","B"; "D","A"; "D","B"; "D
      ","C" }
  Forbidden = { "A","A"; "A","B"; "A","C"; "B","A"; "B","B"; "B","C
      "; "C","C"; "C","D"; "D","D" }
  Reachable = { "A"; "B"; "C"; "D" }
  Root = "A"
}
```

Note that the model is written in the syntactic format of a structure component. The procedures modelexpand and printmodels are only two out of many predefined procedures in IDP. Many procedures provide other forms of inference that can perform computational tasks using the knowledge represented by an FO(ID,AGG,PF,T) theory. Other procedures serve to manipulate and create new logical components, such as vocabularies, structures and theories. We refer to the online IDP Web-IDE (http://dtai.cs.kuleuven.be/krr/idp-ide/) and the IDP manual (https://dtai.cs.kuleuven.be/krr/files/bib/manuals/idp3-manual.pdf) for examples and details. The main design philosophy is that all components in an IDP program are first class citizens. They can be used to perform various reasoning tasks but can also be manipulated by Lua code to construct different ones. So, it is possible to set up a complete workflow.

This methodology in which fine-grained declarative computation steps are mixed in procedures is an exciting novel way of integrating declarative and procedural knowledge. This is illustrated by Bruynooghe et al. [Bruynooghe et al. 2015]. This is an application in stemmatology, the study of the family relationships between different manuscripts (hand made copies) of a text. It sketches a set of procedures that describe the workflow to analyse a number of texts. For each text, a data set is read and analyzed and transformed into structures and vocabularies. These are then combined with the problem vocabulary and theory and, for each so

called feature in the input data of a text, it is checked whether a model exists. Finally, for each data set, a summary report about all its features is reported.

### 1.4.3 Term

One of the available forms of inference in IDP is to solve minimization problems. This inference method takes as input a theory, a partial structure, and a cost term, and outputs one or more models of the theory expanding the partial structure such that the value of the cost term in these structures is minimal (among the set of all models more precise than the partial structure). The cost term is to be specified as a separate term component over the same vocabulary as the theory. To illustrate, we return to our graph problem and introduce a term to count the number of edges in the solution.

```
term t: V{
  #{x y: Edge(x,y)}
}

procedure main(){
  stdoptions.nbmodels = 0
  models, optimal, cost = minimize(T,S,t)
  printmodels(models)
  print(optimal)
  print(cost)
}
```

The main procedure now calls the minimization inference with the theory T, the structure S and the optimization term t as input. With T and S as above, the procedure returns three models, however, it also returns two other values: whether optimality has been proven and the value of the term in the optimal solution. The assignment models, optimal, cost =... assigns them to different variables. The first value is an array of models, which can be printed with printmodels , the other two are simple values; they can be printed with the standard print command. In this example, optimality is reached with a cost of 3.

## 1.5 Advanced Features

### 1.5.1 Constructed Types

In Prolog and ASP, functions and constants have a fixed interpretation, their Herbrand interpretation. Equivalently, we can think of them as logics with built-in *unique names* and *domain closure* axioms (UNA and DCA), i.e., axioms stating that all those values are different and that the domain consists of nothing more than those values, respectively. In FO(ID,AGG,PF,T), the axioms are not present and interpretation of functions and constants is open as in standard FO. Both approaches have their merits, making it useful to integrate the advantages of both. In ASP, there is work to incorporate open functions [Bartholomew and Lee 2012, Lifschitz

2012]. In FO(ID,AGG,PF,T), one way to impose UNA and DCA is to explicitly specify a Herbrand interpretation. This method is illustrated below for the type of days of the week, using the following declarations in the vocabulary:

```
Type Day
monday: Day
tuesday: Day
...
```

and in the structure

```
Day = {''monday''; ''tuesday'' ; ...}
monday = ''monday''
tuesday = ''tuesday''
...
```

There is a way to avoid this cumbersome approach. The following constructed type declaration in the vocabulary expresses the same information but in a compact way:

```
type Day constructed from {monday, tuesday, wednesday, thursday,
    friday}
```

This statement declares several things at once: it declares the type Day and seven constants of this type, it specifies the values for this type and for all its constants in every structure of the vocabulary. Each constant is interpreted by itself. Here, constants and domain elements coincide.

Also non-constant constructor symbols are supported this way. For example, having types row and column, one can introduce the constructed type of positions on a chessboard by declaring "**type** position **constructed from** pos(row,col)", and use it in the definition of a unary predicate queen(position) representing the positions where a queen stands. In theory, this approach also works for recursive constructors and types such as list of integers: **type** list **constructed from** {nil ; cons[**int**, list]}. However, this creates an infinite type and the current IDP solver cannot cope with such types.

### 1.5.2 Structuring Components

All components, vocabularies, theories, structures, terms, and procedures, as we have shown so far, are part of the implicit global namespace idpglobal. This namespace also contains all Lua procedures that are available to the user of the system. When working on large projects, different people may work on different parts, each introducing its own components. To integrate such different parts, the IDP system provides *namespaces*. A namespace with name MySpace is declared by

```
namespace MySpace {
//   content   of   the   namespace
```

```
}
```

A namespace can contain other namespaces, and any sort of IDP component including vocabularies, theories, structures, terms, and procedures. Each component has a full name that is determined by the hierarchy of namespaces it belongs to. This allows users to disambiguate components with the same name but belonging to different namespaces. We refer to the manual for details.

Another useful structuring method is to compose a vocabulary from existing ones. For instance, in the following example W is composed of the symbols of V and one function coloring/1:1 from vocabulary U:[10]

```
vocabulary W {
extern  vocabulary V
extern  U::coloring/1:1
}
```

Such an extension construct is not available for structures and theories but it could be simulated for them by making use of certain Lua procedures. For this we refer to the list of Lua procedures described in the manual.

Worth mentioning is that there is a factlist component to initialize a two valued structure with Prolog or ASP facts. Also, it is possible to call upon Lua procedures to initialize a structure.

### 1.5.3 An Output Vocabulary

In many problems we are interested only in the values of some subset of symbols. In case multiple solutions are searched, we are interested only in models having different interpretations of the output symbols. This is achieved by declaring an output vocabulary, say Vout, and adding it as an extra parameter to the modelexpand call:

```
procedure main(){
    print(modelexpand(T,S,Vout)[1])
}
```

### 1.5.4 Inference Methods

So far we have mentioned *model expansion* inference, invoked as modelexpand(T,S) (or modelexpand(T,S,Vout) if there is an output vocabulary) and *optimization* inference, invoked as minimize(T,S,t) (or minimize(T,S,t,Vout)). The system supports several other inference methods. We discuss the most important ones. For a complete list, we refer to the manual.

[10] Here, the notation coloring/1:1 means that coloring has arity one (/1) and is a function, i.e., has one output argument (:1).

**Query inference** takes as input a *query* component that declares a set expression of the form $\{x \mid \varphi\}$ and a two-valued structure $I$ and returns the set $\{x \mid \varphi\}^I$. In the IDP language, this inference is invoked as query(Q,S), where Q is a query and S a structure. Continuing our graph example,

```
query Q:V{
    {x: (?y: Edge(x,y))}
}

procedure main(){
   models =modelexpand(T,S)
   print(query(Q,models[1]))
}
```

will print the set of all nodes that participate as the first node of an edge in the first model computed by the model expansion. Note that query( ...) does not return a structure but a set.

**Model checking** and **satisfiability checking** are special cases of model expansion. In the former, the input structure is two-valued; the result of this inference is *true* respectively *false* if the input structure is a model of the theory. The latter also outputs a Boolean value, *true* if the (possibly three-valued) output structure *can be expanded* to a model. In the IDP language, both of these inference methods are called using sat(T,S), where T is a theory and S a structure.[11]

**Propagation** inference takes a theory and a structure and returns a more precise structure that preserves all solutions. The system supports different versions of propagation with different costs. The most precise and most expensive version returns the partial structure in which atoms are unknown iff they do not have the same truth value in all models. This most expensive propagation is called using optimalpropagate(T,S). Cheaper, approximate forms of propagation are called using propagate(T,S) and groundpropagate(T,S).

**Deduction** takes as input an FO(ID,AGG,PF,T) theory $\mathcal{T}$ and an FO theory $\mathcal{T}_{FO}$ and returns true if $\mathcal{T} \models \mathcal{T}_{FO}$, that is if the first theory logically entails the second one. It is implemented in a sound but incomplete way by translating $\mathcal{T}$ into a (weaker) FO theory and calling the theorem prover SPASS [Weidenbach et al. 2009]. It is used internally in the IDP system to detect and exploit functional dependencies in predicates [De Cat and Bruynooghe 2013]. It is called using entails(T1,T2).

**Symmetry detection** takes as input a theory $\mathcal{T}$ and a partial structure $I$ and returns *symmetries* over $\mathcal{T}$ and $I$. A symmetry is a function, say $f$, mapping structures to structures, such that, for any two-valued expansion $J$ of $I$ that is a model of $\mathcal{T}$, $f(J)$ is also a model [Devriendt et al. 2012]. Symmetry detection also returns clauses to break these symmetries and to eliminate symmetric models. Symmetry detection is not available as a Lua procedure

[11] Note that satisfiability checking reduces to model checking in case S is two-valued.

but can be exploited in the model expansion workflow using the option symmetrybreaking (see Section 1.6).

**Δ-model expansion** takes as input a definition Δ and a structure $I_{in}$, interpreting all parameters of Δ, and returns the unique model $I$ that expands $I_{in}$. This task is an instance of model expansion, but is solved in IDP using different technology. The close relationship between definitions and logic programs under the well-founded semantics is exploited to translate Δ and $I_{in}$ into a tabled Prolog program, after which XSB is used to compute $I$. Taking an extra formula φ as input, with free variables $\bar{x}$, the same approach is used to solve the query φ with respect to Δ and $I_{in}$ in a goal-oriented way [Jansen et al. 2013]. There is no dedicated Lua procedure for calling Δ-model expansion. However, as described in Section 1.6, it is automatically detected that Δ-model expansion can be performed in normal model expansion calls.

**Unsat-core extraction** takes as input a theory $\mathcal{T}$ and a structure $I$ such that $\mathcal{T}$ has no models expanding $I$. It returns a (minimal) theory $\mathcal{T}_{out}$ entailed by $\mathcal{T}$ (obtained by instantiating some variables) such that $\mathcal{T}_{out}$ still has no models expanding $I$. Currently there is only support for printing the output theory, not for actually obtaining it; this procedures also prints from which line every sentence in $\mathcal{T}_{out}$ was instantiated. This inference is particularly useful when debugging logical specification and can be called using printunsatcore(T ,S).

Finally, there is support for the *linear time calculus* (LTC) defined by Bogaerts et al. [Bogaerts et al. 2014a]. One can build an LTCvocabulary component as a vocabulary extending a default LTC vocabulary and can use special inference methods to initialize the state and to perform progression inference, i.e., to infer the successor states step by step [Bogaerts et al. 2014a].

## 1.6 Under the Hood

In this section we focus on the inner working of some components of the IDP system. First, we discuss the workflow of the optimization inference and how users can control the various parts of the optimization engine. Afterwards we discuss techniques (under development) that help IDP scale to larger, possibly infinite, domains.

Optimization is the task of given a theory $\mathcal{T}$, a structure $I$, and a term $t$, all over the same vocabulary $V$, finding models of $\mathcal{T}$ that expand (are more precise than) $I$. This inference captures Herbrand model generation and (bounded) model expansion, both of which were proposed as logic-based methods for constraint solving, respectively in [East and Truszczyński 2006] and [Mitchell and Ternovska 2005]. In its most general form, we define optimization for typed FO(ID,AGG,PF) as follows. The inference $OPT\langle V, \mathcal{T}, I, t, V_{out}\rangle$ takes as input a theory $\mathcal{T}$, structure $I$ and term $t$, all over vocabulary $V$, and a vocabulary $V_{out} \subseteq V$. Both $\mathcal{T}$ and $I$ are well-typed and $I$ interprets all types. The inference returns $V_{out}$-structures $\mathcal{J}$ such that at least one model of $\mathcal{T}$ expanding both $I$ and $\mathcal{J}$ exists and that expansion is minimal with respect to $t$. The optimization inference is a generalization of the model expansion inference

that takes the same arguments without the optimization term $t$ and that returns $V_{out}$-structures $\mathcal{J}$ such that at least one model of $\mathcal{T}$ expanding both $I$ and $\mathcal{J}$ exists. The workflow of these two inference methods coincides; for optimization, more search is needed to find *optimal* models.

One approach to optimization, used in IDP, is through *ground-and-solve*: ground the input theory and term over the input structure and afterwards apply a search algorithm that, e.g., uses branch-and-bound to find optimal models.

In the rest of the section, we present how the optimization algorithm in IDP solves an $OPT\langle V, \mathcal{T}, I, t, V_{out}\rangle$ task. The workflow consists of an FO(ID,AGG,PF,T) grounding algorithm, a search algorithm for the full ground fragment of FO(ID,AGG,PF,T) and various analysis methods and transformations, that result in a smaller grounding and/or improved search performance.

The workflow of the optimization inference consists of three parts. First, theory and structure are preprocessed to optimize performance. Secondly, the theory is grounded into ECNF, the language supported by MINISAT(ID). Last, the solver MINISAT(ID) is called to perform the actual inference on the ground theory.

### 1.6.1 Preprocessing

Several preprocessing steps are performed before the grounding phase. We briefly discuss them below.

#### 1.6.1.1 Checking structure consistency

First, we specify the structure $I$ that is parsed from a structure component, in particular, how the type autocompletion works. The following definitions formalize this. The value of all other symbols is clear.

We say that a type $t$ of a vocabulary $V$ is explicitly defined in structure $I$ if $I$ contains an equation $t = S$. We define the value of an explicitly defined type in $I$ as stated in its explicit definition. For a type $t$ that is not explicitly defined, the interpretation of $t$ in $I$ consists of all elements of its subtypes and domain elements that occur in the interpretation of symbols $\sigma$ at an argument position of type $t$. Formally:

$$t^I = \left( \bigcup_{\{s | s \text{ is a subtype of } t\}} s^I \right) \bigcup \left( \bigcup_{\{\sigma \in V | \text{the } i\text{'th type of } \sigma \text{ is } t\}} \{d_i \mid \overline{d} \in \sigma^I\} \right).$$

A number of constraints are imposed on structures:

- For any subtype $s$ of $t$, the interpretation of $s$ must be a subset of the interpretation of $t$:

$$\forall x : s(x) \Rightarrow t(x)$$

- If $f$ is a partial or total function that is totally defined, there is at most one image for each value of the input:

$$\forall \bar{x}, y, z : f(\bar{x}) = y \wedge f(\bar{x}) = z \Rightarrow y = z$$

- If $f$ is a totally defined total function, the structure must contain an image for each input argument:

$$\forall \bar{x} : \exists y : f(\bar{x}) = y.$$

- A partially defined predicate cannot be both certainly true and certainly false for the same tuple:

$$\forall \bar{x} : \neg P\langle ct\rangle(\bar{x}) \vee \neg P\langle cf\rangle(\bar{x})$$

- For any partially defined function $f$,
  - there is at most one certainly true image for every input:

$$\forall \bar{x}, y, z : f\langle ct\rangle(\bar{x}, y) \wedge f\langle ct\rangle(\bar{x}, z) \Rightarrow y = z$$

  - if $f$ is a total function, at least one output is possible (not certainly false) for each input:

$$\forall \bar{x} : \exists y : \neg f\langle cf\rangle(\bar{x}, y)$$

When any of these constraints are violated in the autocompleted structure $I$, an appropriate error message is given.

#### 1.6.1.2 Exploiting input$*$-definitions

[Jansen et al. 2013].

Assume that after preprocessing, we obtained a theory $\mathcal{T}$ and a structure $I_1 = I$. The next step is to eliminate some definitions of $\mathcal{T}$ and extend $I_1$. The definitions that can be eliminated are the so called *input$*$-definitions* of $\mathcal{T}$ [Jansen et al. 2013].

We define inductively that a definition $\Delta$ of $\mathcal{T}$ is an input$*$-definition of $\mathcal{T}$ in structure $I$ if all parameters of $\Delta$ have a 2-valued interpretation in $I$ or are defined in input$*$-definitions of $\mathcal{T}$ in $I$.

All input$*$-definitions of $\mathcal{T}$ can be evaluated in advance. Essentially this is done by iterated $\Delta$-model expansion steps: at each iteration an input$*$-definition $\Delta$ is selected that has all its parameters interpreted in the current structure; we compute[12] its model in $I_1$ and add the interpretation of the defined symbols to $I_1$. The advantage of this is that as explained below, top-down grounding techniques, as used in IDP, tend to be rather inefficient in case

[12] This is done by translating it to a logic program and using XSB.

of complex (inductive) definitions [Wittocx 2010]. By evaluating these definitions, we avoid grounding them and we make the input structure more precise.

Notice that this may result in inconsistency if the same predicate is defined in multiple input$*$-definitions that do not agree on its value. Otherwise, the result is a theory $\mathcal{T}_2$ and a refined structure $I_2$.

To exploit the above procedure for a maximal effect, it is extended with a preprocessing step to split each definition $\Delta$ of $\mathcal{T}$ in subdefinitions $\Delta_1, \ldots, \Delta_n$. The advantage is that some of these components may turn out to be input$*$-definitions whereas $\Delta$ is not. In this case we can evaluate part of $\Delta$.

The split of a definition is computed from its dependency relation. Formally, the dependency relation $\leq$ of a definition $\Delta$ is the least transitive relation containing all pairs $P \leq Q$ such that defined symbol $P$ occurs in a rule defining $Q$. We say that $P \sim Q$ if $P \leq Q \leq P$. This is an equivalence relation. The split of $\Delta$ is the partition $\Delta_1, \ldots, \Delta_n$ of $\Delta$ such that each $\Delta_i$ defines an equivalence class of $\sim$. The idea is that each $\Delta_i$ defines a group of predicates that depend on each other.

#### 1.6.1.3 Delaying output$*$-definitions

[Bogaerts et al. 2014b, Jansen et al. 2013] Consider a total definition $\Delta \in \mathcal{T}_2$ such that the defined symbols of $\Delta$ occur only in $\Delta$ and are not interpreted in $I_2$. Any structure that satisfies $\mathcal{T}_2 \setminus \{\Delta\}$ and does not interpret symbols defined in $\Delta$, can be extended to a model of $\mathcal{T}_2$ by evaluating $\Delta$. Consequently, there is no need to consider such a $\Delta$ during search; we prefer to delay evaluation of $\Delta$ as long as possible, to a postprocessing step.

Such a $\Delta$ is one example of an output$*$-definition [Bogaerts et al. 2014b, Jansen et al. 2013]. In general, we define inductively that a definition $\Delta$ of $\mathcal{T}$ is an *output$*$-definition* of $\mathcal{T}$ in structure $I$ if all defined symbols of $\Delta$ only occur in $\Delta$ and in the bodies of rules of output$*$-definitions.

These output$*$-definitions do not have to be considered during search and can be evaluated afterwards in a post-processing step. Theory $\mathcal{T}_3$ is the theory obtained from $\mathcal{T}_2$ by removing all output$*$-definitions; this phase does not modify the structure, hence $I_3 = I_2$.

#### 1.6.1.4 Reducing quantification depth using functional dependencies

[De Cat and Bruynooghe 2013]. The size of the grounding is in general exponential in the nesting depth of quantifiers (as it involves the Cartesian product of the involved domain sizes). One way to reduce the quantification depth, is to detect that symbols can be split into a number of symbols with a smaller arity. Assume, for example, that a predicate timeOf(session ,time) specifies at which time a certain session takes place in a scheduling application. If one could detect that the second argument *functionally depends* on the first argument (the first uniquely determines the value of the second), then it could be replaced by a new function timeFunc(session): time instead. With appropriate transformations, a subformula ?t

: timeOf(s1,t) & timeOf(s2,t) can then be reduced to timeFunc(s1)=timeFunc(s2), eliminating the quantification over t. Detection of functional dependencies is done using the deduction inference: we check whether an FO formula that expresses the dependency is entailed by the original theory. For timeOf, the functional dependency holds if the theory entails the sentence $\forall s : \exists_{=1} t : timeOf(s,t)$.

This preprocessing phase takes as input the theory $\mathcal{T}_3$ and structure $I_3$ and returns $\mathcal{T}_4$ and $I_4$, in which entailed functional dependencies have been made explicit and quantifications have been dropped where possible.

However, as the user expects models in the original vocabulary, additional output*-definitions are added to $\mathcal{T}_4$, that define the original symbol in terms of the newly introduced ones. In our example, this would be the definition

```
define {
  ! s t: timeOf(s,t) <- t = timeFunc(s).
}
```

#### 1.6.1.5 Exploiting symmetries

[Devriendt et al. 2012, 2016b]. It is well-known that if a problem exhibits symmetries, they can cause a search algorithm to solve the same (sub)problem over and over again. For example the "pigeonhole" problem "do $n$ pigeons fit in $n-1$ holes?" is known to be hard for SAT-solvers. Symmetries can be detected and broken on the propositional level [Aloul et al. 2006, Devriendt et al. 2016b], but for large problems, even the task of detecting symmetries becomes infeasible. Detecting symmetries on the first-order theory [Devriendt et al. 2016a] is often an easier problem, as much more structure of the problem is explicitly available. For example for an FO specification of the pigeonhole problem, it is almost trivial to detect that all pigeons are interchangeable. The symmetry detection inference in IDP detects a simple, frequently occurring form of symmetries: locally interchangeable domain elements (see [Devriendt et al. 2016a]). Two domain elements are considered interchangeable if they are of the same type and occur only symmetrically in interpreted predicates. Detected symmetries are handled by adding sentences to $\mathcal{T}_4$ that statically break those symmetries, resulting in the theory $\mathcal{T}_5$.

### 1.6.2 Ground-And-Solve

#### 1.6.2.1 Ground

[De Cat et al. 2013]. The grounding algorithm visits the resulting theory ($\mathcal{T}_5$) in a depth-first, top-down fashion, basically replacing all variables by all their matching instantiations, according to the interpretation of their types in a partial structure $I_5$. For example, a formula $\forall x[t] : \psi(x)$ is replaced by $\bigwedge_{d \in t^{I_5}} \psi(d)$.

However, such an instantiation might be unnecessary large. Indeed, if the value of a term or formula is known in the current structure $I_5$ for a given instantiation of its free variables, it

should not have been grounded in the first place. The solution is to reuse query inference. For example, consider a formula $\forall \overline{x}[\overline{T}] : \varphi$ and an instantiation $\overline{d}$ for the free variables $\overline{y}$ of this formula. In that case, $\overline{x}$ need only be instantiated with tuples $\overline{d}'$ for which $\varphi[\overline{y}/\overline{d},\overline{x}/\overline{d}']$ is not certainly true in $I_5$. Finding such tuples can be done using the query inference on a derived structure over a vocabulary in which <ct> and <cf> tables have an explicit representation. For all instantiations of $\overline{x}$ not in the result of that query, we are certain that the subformula is true anyway. In fact, an incomplete (cheaper) query inference can be applied, as any over-approximation will result in additional grounding, still maintaining correctness. The result is a ground FO(ID,AGG,PF,T) theory. Several optimizations for this step exist, as discussed in [Jansen et al. 2014]; below we discuss some of them.

#### 1.6.2.2 Simplification

To ensure models are generated that expand the input structure, not only the ground theory, but also the structure $I_5$ is passed to the search algorithm. Since we use a top-down grounding algorithm, we can optimize over this: whenever a domain atom or term is generated by instantiating variables, instead of using the atom or term itself, its interpretation is filled in in the grounding. For example, if a formula $\forall x[t] : P(x) \vee Q(x)$ is grounded in a structure with interpretations

```
T={1;2;3}
P<ct> = {1}
Q<cf> = {3}
```

a simplified grounding is

$$(\mathbf{t} \vee Q(1)) \wedge (P(2) \vee Q(2)) \wedge (P(3) \vee \mathbf{f}).$$

This sentence can be simplified even more, by propagating derived truth values upwards, resulting in

$$\mathbf{t} \wedge (P(2) \vee Q(2)) \wedge P(3)$$

and finally

$$(P(2) \vee Q(2)) \wedge P(3).$$

These simplification techniques can have as effect that large parts of the theory do not need to be grounded. For example, consider a sentence $\forall x[t] : P(x) \vee \varphi(x)$, where $\varphi$ might be a large formula. For all instantiations $d$ of $x$ for which $P(d)$ holds, the formula $P(d) \vee \varphi(d)$ simplifies to $\mathbf{t}$. Hence $\varphi(d)$ does not need to be grounded for such $d$.

#### 1.6.2.3 Approximation and lifted unit propagation

[Wittocx et al. 2010, 2013].

The above grounding algorithm exploits information in the input structure $I$ using the query inference. Essentially, it grounds only the instances $\varphi[\overline{d}]$ of formulas that are unknown in $I$. As a consequence, more precise input structures $I$ yield smaller groundings and increased search performance. This observation gave rise to the algorithms presented in [Wittocx et al. 2010], where instead of using structure $I_5$ directly, we first compute a more precise structure $I_6$ that approximates all models of $\mathcal{T}_5$ that expand $I_5$. Ideally, we would like to compute the *most precise* structure that is less precise than all models of $\mathcal{T}_5$ that expand $I_5$. Of course, finding this ideal structure is a task that is even harder than the original problem.

Instead of searching for this ideal structure, IDP's approach is to execute a lifted (approximative) version of the unit propagation that would occur after grounding. The result is stored as a symbolic representation of a structure. Namely, with each symbol $P$, we associate two symbolic set expressions $S_{ct}$ and $S_{cf}$ with intended meaning that in structure $I_6$, $P_{ct}$, respectively $P_{cf}$, is interpreted as $S_{ct}^{I_5}$, respectively $S_{cf}^{I_5}$. Consider, for example, the following theory

$$\forall x : P(x) \Rightarrow Q(x).$$
$$\forall x : \neg Q(x) \Rightarrow R(x).$$

Symbolic unit propagation then results in, e.g., a symbolic representation of $I_6$ that interprets $Q_{ct}$ as $\{x \mid P_{ct}(x) \vee R_{cf}(x)\}$ (the latter interpreted in $I_5$!), $R_{ct}$ as $\{x \mid Q_{cf}(x)\}$. During the grounding phase, all queries for variable instantiations and the interpretation of atoms and terms are evaluated relative to this symbolic interpretation, resulting in fewer instantiations and more precise interpretations. E.g., if $P$ is interpreted in $I_5$, and $Q$ and $R$ are completely unknown in $I_5$, then the second sentence will only be instantiated for $x$'s such that that $P(x)$ is not true in $I_5$.

A symbolic representation of complete lifted unit presentation often consists of complex formulas, which are infeasible to query. However, any approximation of those formulas is sufficient, as long as the resulting structure is at least as precise as $I_5$. Consequently, the formulas are simplified to balance the estimated cost of querying against the expected reduction in number of answers.

#### 1.6.2.4 Search

Optimization in IDP relies on the search algorithm MINISAT(ID) [De Cat et al. 2013] for ground FO(ID,AGG,PF,T) theories. It takes the ground theory as input together with structure $I_5$. The algorithm combines techniques from SAT, Constraint Programming (CP) and Answer Set Programming (ASP) through a DPLL(T) architecture [Ganzinger et al. 2004]. At the core lies the SAT-solver MINISAT [Eén and Sörensson 2003], a complete, Boolean search algorithm for propositional clauses. This core is complemented by a range of "propagator" modules that take care of propagation for all other types of constraints in the theory, such as aggregates, definitions and atoms containing functions. Each module is

responsible for explaining its propagations in terms of the current assignment. For a definition $\Delta$, for example, the module checks whether the current assignment satisfies $\Delta$'s completion, whether the current assignment contains unfounded sets, and when a complete assignment is found, whether the structure is the well-founded model of $\Delta$. Optimization is taken care of by a module that ensures the search space is visited in a branch-and-bound fashion. Whenever a model $M$ is found, with value $v$ the interpretation of $c$ in $M$, a constraint $c < v$ is added to the ground theory (which raises a conflict, leading to backtracking and additional search).

***On the importance of CP integration*** [De Cat et al. 2013, 2014] In contrast to previous versions, the current version of MINISAT(ID) supports ground FO(ID,AGG,PF,T) *with function symbols*. Function symbols are handled using techniques from constraint programming [Feydy and Stuckey 2009]. To illustrate the importance of having uninterpreted function symbols, consider the following birthday riddle.

**Example 1.6.1.** "To determine my age, it suffices to know that my age in 2013 is halfway between two consecutive primes, that my age's prime factors do not sum to a prime number, and that I was born in a prime year.". In the IDP language, it can be modeled as:

```
vocabulary V {
    type Nb isa nat
    Age: Nb          //uninterpreted constant: my age
    Prime(Nb)        //predicate containing all prime numbers
    YearOfBirth: Nb //uninterpreted constant: my year of birth
}
theory T : V  {
    //Definition of prime numbers
    define {
        ! x[Nb]: Prime(x) <-
            x>1 &
            !y [Nb]: 1 < y < x =>  (x % y ~= 0).
    }

    //Relation between age (in 2013) and year of birth
    Age = 2013 - YearOfBirth.

    //My age in 2013 is halfway between two consecutive primes
    ?x1 x2:
        //x1 and x2 are prime
        Prime(x1) & Prime(x2) &
        //they are consecutive
        ~(?y: Prime(y) & x1 < y < x2) & x1 < x2 &
        // my age is halfway between them
        Age = (x2 + x1)/2.
```

```
    //I was born in a prime year
    Prime(YearOfBirth).

    //my age's prime factors do not sum to a prime number
    ~Prime(sum{x: Prime(x) & 1 < x =< Age & Age % x = 0 : x}).
}

structure S : V {
    Nb = {0..2013}
}
```

IDP is unable to ground this theory without using uninterpreted function symbols (this is controlled through the option cpsupport, discussed below) due to memory exhaustion. With uninterpreted constants, IDP takes half a second to find a solution. In fact, IDP proves that 48 different solutions exist; however only one is an age below 100, namely $Age = 26$.

### 1.6.3 Post-Processing

As a final, post-processing step, structures returned by the search step are translated back to structures over $V$. Next, they are merged with $I$, output$*$-definitions are evaluated over them and finally, they are projected to $V_{out}$, resulting in structures $I_{out}$ that are solutions to the original $OPT\langle V, \mathcal{T}, I, t, V_{out}\rangle$ problem.

An output$*$-definition is only evaluated if evaluating it will have an effect on the eventual $V_{out}$-structure. If symmetry-breaking was applied, additional solutions can be generated by applying the symmetries to the solutions found.

### 1.6.4 Controlling the optimization workflow

Various components of the optimization workflow can be controlled using options. We provide a brief overview.

**stdoptions.assumeconsistentinput** *default:false*

If this option is true, the systems assumes that the input provided by the user is consistent and the consistency checks are skipped. Use at your own risk.

**stdoptions.xsb** *default:true*

If this option is true, input$*$-definitions are evaluated using the XSB Prolog system. Otherwise, they are evaluated using standard ground-and-solve techniques. We recommend to use XSB for efficiency reasons.

**stdoptions.postprocessdefs** *default:true*

This option controls whether output$*$-definitions are delayed until after search. In general, we recommend to turn this option on. However, since detection of output$*$-definitions is implemented through a bootstrapping approach, enabling might cause a

small delay of up to a second. Hence, if your goal is to solve a large number of very small problems, we recommend to turn this off.

**stdoptions.splitdefs** *default:true*

This option controls whether definitions are split into minimal strata. As with the previous, we recommend to turn it on in most cases, unless when solving very small problems.

**stdoptions.symmetrybreaking** *default:"none"*

This option controls whether symmetries are broken and if they are broken, using which method. Currently, IDP only provides support for breaking symmetries *statically*, hence this option can be set either to ''none'' or to '' static ''. Future versions of IDP might also offer the choice ''dynamic''. Whether symmetry breaking is beneficial strongly depends on the problem at hand.

**stdoptions.reducedgrounding** *default:true*

This option controls whether the grounding is simplified using information from the structure. We recommend to enable this option in most use cases.

**stdoptions.groundwithbounds** *default:true*

This option controls whether the grounding size is reduced using the approximation techniques described above. We recommend enabling this option in most use cases.

**stdoptions.liftedunitpropagation** *default:true*

This option controls whether the symbolic representation of the input structure ($I_6$) is evaluated in advance, resulting in a concrete representation of the input structure. We recommend enabling this option in most use cases.

**stdoptions.cpsupport** *default:true*

This option controls whether function symbols are allowed in the grounding. If turned off, the ground theory will be entirely propositional, if turned on, functions symbols can appear in the ground theory; they are then handled by constraint programming techniques. We recommend enabling this option in most use cases, except for hard combinatorial problems with a very small grounding.

**stdoptions.cpgroundatoms** *default:false*

This option controls whether function symbols are allowed to occur *nested* in the grounding. This is an advanced feature, with as advantage a smaller grounding, but as disadvantage possible loss of propagation. Whether enabling this option is beneficial strongly depends on the problem at hand.

**stdoptions.functiondetection** *default:false*

This option controls whether predicate symbols are automatically replaced by function symbols. Whether enabling this option is beneficial depends strongly on the problem at

hand. For theories crafted by experts (and manually optimised), this options is probably not beneficial. The more naive the theory is, the more potential benefit this option has. If this option is enabled, we recommend to also enable the cpsupport option.

**stdoptions.nbmodels** *default:1*

This option specifies the number of models that need to be returned by the optimization (or model expansion) inference.

**stdoptions.verbosity** *default:0 (for all suboptions)*

Suboptions of this option control the verbosity of various components in the workflow. This is used mainly for debugging purposes, e.g., to know which part of the solver is causing certain delays.

### 1.6.5 Scalability and Infinity

The reader might have noticed that structures and groundings can be very large or even infinite (for example, when a predicate or a quantified variable is typed over **int**). Model expansion (and thus, optimization) over infinite structures takes infinite time in general. In IDP, several techniques are applied to address this issue; they often work well in practice.

A first such technique has been explained in Subsection 1.6.2. By intelligent reasoning over the entire theory, we can sometimes derive better variable bounds. Suppose for example that a theory contains formulas $\forall x[int] : P(x) \Rightarrow Q(x)$ and $\forall y[int] : P(y) \Rightarrow R(y)$, where $Q$ only ranges over a finite type, say $T$, but $P$ and $R$ range over $int$. The first of these sentences guarantees that $P$ will only hold for values such that $Q$ holds, hence $P$ can only hold for values in the finite type $T$. Thus we know that the second sentence should only be instantiated for $y$'s in $T$, i.e., by deriving an improved bound for $y$, the grounding of the second sentence suddenly becomes finite. The first sentence can be handled similarly. We only ground this sentence for $y$'s in $T$ and maintain a symbolic interpretation expressing that $P$ is certainly false outside of $T$.

Second, the usage of a top-down, depth-first grounding algorithm has the advantage that interpretations can be evaluated *lazily*: (i) instantiations of quantifications can be generated one at a time, and (ii) the interpretation of atoms and terms needs only to be retrieved for atoms and terms that effectively occur in the grounding. The same advantage applies for symbols that are interpreted by (complex) procedures: the procedures are only executed for relevant occurrences of that symbol.

Third, the search algorithm maintains bounds on the interpretation of function terms, taking constraints in the grounding into account. Consider a constant $c : int$, which in itself would result in an infinite search space. However, combined with, e.g., a constraint $0 \leq c \leq 10$ in the grounding, the solver reduces $c : int$ to $c : [0, 10]$, a finite search space.

A fourth technique currently under development to increase scalability is *Lazy Grounding* [De Cat et al. 2015]. Lazy Grounding is based on the observation that the entire grounding

often is not necessary to find solutions to a model expansion or optimisation problem. Instead, the technique interleaves grounding with search as follows. Initially, it (roughly) splits the input theory into two parts: one part is grounded and the underlying solver performs its standard search algorithm on it. The other part of the theory is *delayed*: the system makes assumptions about it that guarantee that models found by the solver can be extended to models of the entire theory. Whenever these assumptions become violated, i.e., when they are inconsistent with the solver's current assignment, the splitting of the theory is revised. Consider for example $\forall x[int] : P(x) \Rightarrow \varphi$ with $\varphi$ a possibly large formula; it has an infinite grounding. A smart lazy grounder could delay the grounding of that sentence with the assumption that $P$ is false for all integers. During search, only when an atom $P(d)$ becomes true, is the sentence grounded for $x = d$ and only that ground sentence is added to the search, the remainder still delayed on the assumption that $P$ is false. Whenever the search algorithm finds a structure that satisfies the grounding and does not violate any assumptions, that structure can be straightforwardly extended to a model of the whole theory.

## 1.7 In Practice

Both IDP and its search algorithm MINISAT(ID) are open-source systems, freely available from dtai.cs.kuleuven.be/krr/software. Next to accepting input in the IDP language, both systems provide C++ interfaces. The search algorithm MINISAT(ID) supports input in Clausal Normal Form (CNF), Quantified Boolean Form (QBF, CNF's higher-order relative) [Egly et al. 2000], ground ASP (in the LParse-Smodels intermediate format [Syrjänen 1998]) and FlatZinc [Nethercote et al. 2007].

IDP can be be tried online in our web-IDE at https://dtai.cs.kuleuven.be/krr/idp-ide/. Several modeling examples are available in the web-IDE as well as at https://dtai.cs.kuleuven.be/software/idp/examples. The web-IDE also provides support for the visualisations of structures, as explained, e.g., in https://dtai.cs.kuleuven.be/krr/idp-ide/?chapter=intro/9-IDPD3.

The main usage of IDP3 is currently its model expansion inference, which as discussed earlier, is closely related to generating answer sets of logic programs and to solving constraint satisfaction problems. As such, it shares applications with those domains, general examples of which are scheduling, planning, verification and configuration problems. More concretely, some applications have been modelled in [Bruynooghe et al. 2015], demonstrating its applicability as both an approach to replace procedural programming in some cases and as an approach to rapid prototyping due to the short development time. It has been used to analyze security issues in several contexts, with an emphasis on formal approaches that allow intuitive modeling of the involved knowledge [Decroix et al. 2013, Heyman 2013]. The model expansion engine, and various other types of inference, have been used for interactive configuration [Van Hertum et al. 2016, 2017].

The performance of IDP has been demonstrated for example in the ASP competition series, in 2009 [Denecker et al. 2009] (IDP$^2$), 2011 [Calimeri et al. 2014] (IDP$^2$) and 2013 [Alviano

et al. 2013] (IDP3) and in [Bruynooghe et al. 2015], where it is compared to various existing approaches to specific problems. The performance of the search algorithm MINISAT(ID) has been demonstrated in [Amadini et al. 2013a,b], where it turned out to be the single-best solver in their MiniZinc portfolio, and in the latest Minizinc challenges [MinizincChallenge2012]. In [Bruynooghe et al. 2015], it is demonstrated that a KR system like IDP can be practically used as a front-end for lower-level solvers (e.g., SAT-solvers), instead of manually encoding problems in SAT or using custom scripting. Experimental results showed that IDP is able to relieve this burden with minimal performance loss and greatly reduced development effort.

IDP3 is used as a didactic tool in various logic-oriented courses at various universities.

## 1.8 Related Work

Within several domains, research is targeting expressive specification languages and (to a lesser extent) multiple inference techniques within one language. While we do not aim at an extensive survey of related languages (e.g., [Marriott et al. 2008] has a section with such a survey), we do compare with a couple of them.

The B language [Abrial 1996], a successor of Z, is a formal specification language developed specifically for the generation of procedural code. It is based on first-order logic and set theory, and supports quantification over sets. Event-B is a variant for the specification of event-based applications. The language Zinc, developed by Marriott et al. [Marriott et al. 2008], is a successor of OPL and intended as a specification language for constraint programming applications (mainly CSP and COP solving). It is based on first-order logic, type theory and constraint programming languages. Within ASP, a number of related languages, originating from logic programs, are being developed, such as Gringo [Gebser et al. 2009] and DLV [Leone et al. 2006]. They support definitional knowledge and default reasoning. Implementations exist for inference techniques like stable model generation (related to model expansion), visualisation, optimization and debugging. A comparison of ASP and FO(ID) can be found in [Denecker et al. 2012]. The language of the Alloy [Jackson 2002] system is basically first-order logic extended with relational algebra operators, but with an object-oriented syntax, making it more natural to express knowledge from application domains centered around agents and their roles, e.g., security analysis.

The following are alternative approaches to model expansion (or to closely related inference tasks). The solver-independent CP language Zinc [Marriott et al. 2008] is grounded to the language MiniZinc [Nethercote et al. 2007], supported by a range of search algorithms using various paradigms, as can be seen on www.minizinc.org/challenge2012/results2012.html. In the context of constraint ASP (CASP), several systems ground to ASP extended with constraint atoms, such as Clingcon [Ostrowski and Schaub 2012] and EZ(CSP) [Balduccini 2011]. For search, Clingcon combines the ASP solver Clasp [Gebser et al. 2012b] with the CSP solver Gecode [Gecode Team 2013], while EZ(CSP) combines an off-the-shelf ASP solver with an off-the-shelf CLP-Prolog system. The prototype CASP solver Inca [Drescher

and Walsh 2012] searches for answer sets of a ground CASP program by applying Lazy Clause Generation (LCG) for arithmetic and all-different constraints. As opposed to extending the search algorithm, a different approach is to transform a CASP program to a pure ASP program [Drescher and Walsh 2011], afterwards applying any off-the-shelf ASP solver. CASP languages generally only allow a restricted set of expressions to occur in constraint atoms and impose conditions on where constraint atoms can occur. For example, none of the languages allows general atoms $P(\overline{c})$ with $P$ being an uninterpreted predicate symbol. One exception is $\mathcal{AC}(\mathcal{C})$, a language aimed at integrating ASP and Constraint Logic Programming [Mellarkod et al. 2008]. As shown in [Lierler 2012], the language captures the languages of both Clingcon and EZ(CSP); however, only subsets of the language are implemented [Gelfond et al. 2008].

## 1.9 Conclusion

Kowalski's 1974 paper [Kowalski 1974] laid the foundations for the field of Logic Programming, by giving the Horn-clause subset of predicate logic a procedural interpretation to use it for programming. More recently, progress in automated reasoning in fields such as SAT and CP made the exploration possible of more pure forms of declarative programming, gradually moving from declarative programming to declarative modeling, in which the user only has to care about the problem specification.

In this chapter, we took this development one step further and presented the knowledge base system IDP, in which knowledge is separated from computation. The knowledge representation language is both natural and extensible, cleanly integrating first-order logic with definitions, aggregates, etc. It provides a range of inference engines and functionalities for tasks encountered often in practice.

IDP is an extensible framework for declarative modeling, in which both language extensions and inference engines can be added with relative ease. It focuses on moving the burden of performance on modeling from the user to the system, demonstrated by the workflow of optimization inference, which is achieved by combining insights from fields such as SAT, constraint programming, logic programming and answer set programming.

# Bibliography


J.-R. Abrial. 1996. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA. ISBN 0-521-49619-5.

J.-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. 2010. Rodin: An open toolset for modelling and reasoning in Event-B. *STTT*, 12(6): 447–466.

F. A. Aloul, K. A. Sakallah, and I. L. Markov. 2006. Efficient symmetry breaking for Boolean satisfiability. *IEEE Transactions on Computers*, 55(5): 549–558. ISSN 0018-9340. DOI: 10.1109/TC.2006.75.

M. Alviano, F. Calimeri, G. Charwat, M. Dao-Tran, C. Dodaro, G. Ianni, T. Krennwallner, M. Kronegger, J. Oetsch, A. Pfandler, J. Pührer, C. Redl, F. Ricca, P. Schneider, M. Schwengerer, L. K. Spendier, J. P. Wallner, and G. Xiao. 2013. The fourth Answer Set Programming competition: Preliminary report. In P. Cabalar and T. C. Son, eds., *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings*, volume 8148 of *LNCS*, pp. 42–53. Springer. ISBN 978-3-642-40563-1, 978-3-642-40564-8. http://dx.doi.org/10.1007/978-3-642-40564-8_5.

R. Amadini, M. Gabbrielli, and J. Mauro. 2013a. Features for building CSP portfolio solvers. *CoRR*, abs/1308.0227.

R. Amadini, M. Gabbrielli, and J. Mauro. 2013b. An empirical evaluation of portfolios approaches for solving CSPs. In C. P. Gomes and M. Sellmann, eds., *CPAIOR*, volume 7874 of *Lecture Notes in Computer Science*, pp. 316–324. Springer. ISBN 978-3-642-38170-6. http://dx.doi.org/10.1007/978-3-642-38171-3_21. DOI: 10.1007/978-3-642-38171-3_21.

M. Balduccini. 2011. Industrial-size scheduling with ASP+CP. In J. P. Delgrande and W. Faber, eds., *LPNMR*, volume 6645 of *Lecture Notes in Computer Science*, pp. 284–296. Springer. ISBN 978-3-642-20894-2. http://dx.doi.org/10.1007/978-3-642-20895-9_33. DOI: 10.1007/978-3-642-20895-9_33.

M. Bartholomew and J. Lee. 2012. Stable models of formulas with intensional functions. In Brewka et al. [2012]. ISBN 978-1-57735-560-1. http://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4559.

B. Bogaerts, B. De Cat, S. De Pooter, and M. Denecker, 2012. The IDP framework reference manual. https://dtai.cs.kuleuven.be/krr/files/bib/manuals/idp3-manual.pdf.

B. Bogaerts, J. Jansen, M. Bruynooghe, B. De Cat, J. Vennekens, and M. Denecker. 7 2014a. Simulating dynamic systems using linear time calculus theories. *TPLP*, 14(4–5): 477–492. ISSN 1475-3081. http://journals.cambridge.org/article_S1471068414000155. DOI: 10.1017/S1471068414000155.

B. Bogaerts, J. Jansen, B. De Cat, G. Janssens, M. Bruynooghe, and M. Denecker. 2014b. Meta-level representations in the IDP knowledge base system: Towards bootstrapping inference engine development. In D. Mitchell and M. Denecker, eds., *Workshop on Logic and Search, 2014*, pp. 1–14. https://lirias.kuleuven.be/handle/123456789/459071.

G. Brewka, T. Eiter, and S. A. McIlraith, eds. 2012. *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-*

*14, 2012*. AAAI Press. ISBN 978-1-57735-560-1.

M. Bruynooghe, H. Blockeel, B. Bogaerts, B. De Cat, S. De Pooter, J. Jansen, A. Labarre, J. Ramon, M. Denecker, and S. Verwer. November 2015. Predicate logic as a modeling language: modeling and solving some machine learning and data mining problems with IDP3. *TPLP*, 15(6): 783–817. ISSN 1475-3081. http://journals.cambridge.org/article_S147106841400009X. DOI: 10.1017/S147106841400009X.

W. Buchholz, S. Feferman, W. Pohlers, and W. Sieg. 1981. *Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof-Theoretical Studies*, volume 897 of *Lecture Notes in Mathematics*. Springer.

F. Calimeri, G. Ianni, and F. Ricca. 2014. The third open answer set programming competition. *TPLP*, 14(1): 117–135. http://dx.doi.org/10.1017/S1471068412000105. DOI: 10.1017/S1471068412000105.

W. Chen and D. S. Warren. 1996. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1): 20–74.

K. L. Clark. 1978. Negation as failure. In *Logic and Data Bases*, pp. 293–322. Plenum Press. ISBN 0-306-40060-X.

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

B. De Cat, B. Bogaerts, J. Devriendt, and M. Denecker. 2013. Model expansion in the presence of function symbols using constraint programming. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013*, pp. 1068–1075. IEEE Computer Society. ISBN 978-1-4799-2971-9. http://dx.doi.org/10.1109/ICTAI.2013.159. DOI: 10.1109/ICTAI.2013.159.

B. De Cat, B. Bogaerts, and M. Denecker, Sept. 2014. MiniSAT(ID) for satisfiability checking and constraint solving. ALP Newsletter. https://lirias.kuleuven.be/handle/123456789/463884.

B. De Cat, M. Denecker, M. Bruynooghe, and P. J. Stuckey. 2015. Lazy model expansion: Interleaving grounding with search. *J. Artif. Intell. Res. (JAIR)*, 52: 235–286. http://dx.doi.org/10.1613/jair.4591. DOI: 10.1613/jair.4591.

S. De Pooter, J. Wittocx, and M. Denecker. 2011. A prototype of a knowledge-based programming environment. *CoRR*, abs/1108.5667.

K. Decroix, J. Lapon, B. De Decker, and V. Naessens. 2013. A formal approach for inspecting privacy and trust in advanced electronic services. In J. Jürjens, B. Livshits, and R. Scandariato, eds., *ESSoS*, volume 7781 of *LNCS*, pp. 155–170. Springer. ISBN 978-3-642-36562-1.

M. Deinum, J. Long, G. Mak, and D. Rubio. 2014. *Spring Annotation Driven Core Tasks*, pp. 135–216. Apress, Berkeley, CA. ISBN 978-1-4302-5909-1. https://doi.org/10.1007/978-1-4302-5909-1_3. DOI: 10.1007/978-1-4302-5909-1_3.

M. Denecker. 1998. The well-founded semantics is the principle of inductive definition. In J. Dix, L. F. del Cerro, and U. Furbach, eds., *JELIA*, volume 1489 of *LNCS*, pp. 1–16. Springer. ISBN 3-540-65141-1.

M. Denecker. 2000. Extending classical logic with inductive definitions. In J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, eds., *CL*, volume 1861 of *LNCS*, pp. 703–717. Springer. ISBN 3-540-67797-6.

M. Denecker and E. Ternovska. Apr. 2008. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log.*, 9(2): 14:1–14:52. ISSN 1529-3785. http://dx.doi.org/10.1145/1342991.1342998.

M. Denecker and J. Vennekens. 2007. Well-founded semantics and the algebraic theory of nonmonotone inductive definitions. In C. Baral, G. Brewka, and J. S. Schlipf, eds., *LPNMR*, volume 4483 of *Lecture Notes in Computer Science*, pp. 84–96. Springer. ISBN 978-3-540-72199-4. http://dx.doi.org/10.1007/978-3-540-72200-7_9. DOI: 10.1007/978-3-540-72200-7_9.

M. Denecker and J. Vennekens. 2008. Building a knowledge base system for an integration of logic programming and classical logic. In M. García de la Banda and E. Pontelli, eds., *ICLP*, volume 5366 of *LNCS*, pp. 71–76. Springer. ISBN 978-3-540-89981-5. http://dx.doi.org/10.1007/978-3-540-89982-2_12.

M. Denecker and J. Vennekens. 2014. The well-founded semantics is the principle of inductive definition, revisited. In C. Baral, G. De Giacomo, and T. Eiter, eds., *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference*, pp. 1–10. AAAI Press. ISBN 978-1-57735-657-8. http://www.aaai.org/ocs/index.php/KR/KR14/paper/view/7957.

M. Denecker, M. Bruynooghe, and V. Marek. 2001. Logic programming revisited: Logic programs as inductive definitions. *ACM Trans. Comput. Log.*, 2(4): 623–654.

M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczyński. 2009. The second answer set programming competition. In Erdem et al. [2009], pp. 637–654. ISBN 978-3-642-04237-9.

M. Denecker, Y. Lierler, M. Truszczyński, and J. Vennekens. 2012. A Tarskian informal semantics for answer set programming. In Dovier and Costa [2012], pp. 277–289. ISBN 978-3-939897-43-9.

J. Devriendt, B. Bogaerts, B. De Cat, M. Denecker, and C. Mears. 2012. 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*, pp. 49–56. IEEE Computer Society. ISBN 978-1-4799-0227-9. http://dx.doi.org/10.1109/ICTAI.2012.16. DOI: 10.1109/ICTAI.2012.16.

J. Devriendt, B. Bogaerts, M. Bruynooghe, and M. Denecker. 009 2016a. On local domain symmetry for model expansion. *Theory and Practice of Logic Programming*, 16(5-6): 636–652. https://www.cambridge.org/core/article/on-local-domain-symmetry-for-model-expansion/96E8AB07EB4C02D502B68687B23AC21C. DOI: 10.1017/S1471068416000508.

J. Devriendt, B. Bogaerts, M. Bruynooghe, and M. Denecker. 2016b. Improved static symmetry breaking for SAT. In N. Creignou and D. L. Berre, eds., *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pp. 104–122. Springer. ISBN 978-3-319-40969-6. http://dx.doi.org/10.1007/978-3-319-40970-2_8. DOI: 10.1007/978-3-319-40970-2_8.

A. Dovier and V. S. Costa, eds. 2012. *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary. Proceedings*, volume 17 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-43-9.

C. Drescher and T. Walsh. 2011. Translation-based constraint answer set solving. In T. Walsh, ed., *IJCAI*, pp. 2596–2601. IJCAI/AAAI. ISBN 978-1-57735-516-8. http://ijcai.org/papers11/Papers/IJCAI11-432.pdf.

C. Drescher and T. Walsh. 2012. Answer set solving with lazy nogood generation. In Dovier and Costa [2012], pp. 188–200. ISBN 978-3-939897-43-9.

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

N. Eén and N. Sörensson. 2003. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, eds., *SAT*, volume 2919 of *LNCS*, pp. 502–518. Springer. ISBN 3-540-20851-8.

U. Egly, T. Eiter, H. Tompits, and S. Woltran. 2000. Solving advanced reasoning tasks using quantified Boolean formulas. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pp. 417–422. AAAI Press. ISBN 0-262-51112-6.

E. Erdem, F. Lin, and T. Schaub, eds. 2009. *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *LNCS*. Springer. ISBN 978-3-642-04237-9.

T. Feydy and P. J. Stuckey. 2009. Lazy clause generation reengineered. In Gent [2009], pp. 352–366. ISBN 978-3-642-04243-0. http://dx.doi.org/10.1007/978-3-642-04244-7_29. DOI: 10.1007/978-3-642-04244-7_29.

A. M. Frisch and P. J. Stuckey. 2009. The proper treatment of undefinedness in constraint languages. In Gent [2009], pp. 367–382. ISBN 978-3-642-04243-0. http://dx.doi.org/10.1007/978-3-642-04244-7_30. DOI: 10.1007/978-3-642-04244-7_30.

H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. 2004. DPLL( T): fast decision procedures. In R. Alur and D. Peled, eds., *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *LNCS*, pp. 175–188. Springer. ISBN 3-540-22342-8. http://dx.doi.org/10.1007/978-3-540-27813-9_14. DOI: 10.1007/978-3-540-27813-9_14.

M. Gebser, R. Kaminski, M. Ostrowski, T. Schaub, and S. Thiele. 2009. On the input language of ASP grounder Gringo. In Erdem et al. [2009], pp. 502–508. ISBN 978-3-642-04237-9.

M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. 2012a. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers. dx.doi.org/10.2200/S00457ED1V01Y201211AIM019.

M. Gebser, B. Kaufmann, and T. Schaub. 2012b. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187: 52–89.

Gecode Team, 2013. Gecode: Generic constraint development environment. Available from `http://www.gecode.org`.

M. Gelfond and V. Lifschitz. 1988. The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, eds., *ICLP/SLP*, pp. 1070–1080. MIT Press. ISBN 0-262-61056-6. http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.6050.

M. Gelfond and V. Lifschitz. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4): 365–386.

M. Gelfond, V. S. Mellarkod, and Y. Zhang. 2008. Systems integrating answer set programming and constraint programming. In M. Denecker, ed., *Second Workshop on Logic and Search, 2008*, pp. 145–152. ACCO.

I. P. Gent, ed. 2009. *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 of *Lecture Notes in Computer Science*. Springer. ISBN 978-3-642-04243-0. http://dx.doi.org/10.1007/978-3-642-04244-7. DOI: 10.1007/978-3-642-04244-7.

T. Heyman. Mar 2013. *A Formal Analysis Technique for Secure Software Architectures*. PhD thesis, Department of Computer Science, KU Leuven. https://lirias.kuleuven.be/handle/123456789/389365.

R. Ierusalimschy, L. Henrique de Figueiredo, and W. Celes. 1996. Lua – an extensible extension language. *Software: Practice and Experience*, 26(6): 635–652. ISSN 1097-024X. http://dx.doi.org/10.1002/(SICI)1097-024X(199606)26:6⟨635::AID-SPE26⟩3.0.CO;2-P. DOI: 10.1002/(SICI)1097-024X(199606)26:6¡635::AID-SPE26¿3.0.CO;2-P.

D. Jackson. 2002. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM'02)*, 11(2): 256–290.

J. Jansen, A. Jorissen, and G. Janssens. 2013. Compiling input∗ FO(·) inductive definitions into tabled Prolog rules for IDP3. *TPLP*, 13(4–5): 691–704. DOI: 10.1017/S1471068413000434.

J. Jansen, I. Dasseville, J. Devriendt, and G. Janssens. 2014. Experimental evaluation of a state-of-the-art grounder. In O. Chitil, A. King, and O. Danvy, eds., *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, pp. 249–258. ACM. ISBN 978-1-4503-2947-7. http://doi.acm.org/10.1145/2643135.2643149. DOI: 10.1145/2643135.2643149.

R. A. Kowalski. 1974. Predicate logic as programming language. In *IFIP Congress*, pp. 569–574.

N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3): 499–562.

Y. Lierler. 2012. On the relation of constraint answer set programming languages and algorithms. In J. Hoffmann and B. Selman, eds., *AAAI*. AAAI Press.

V. Lifschitz. 2012. Logic programs with intensional functions. In Brewka et al. [2012]. ISBN 978-1-57735-560-1.

J. W. Lloyd. 1987. *Foundations of Logic Programming, 2nd Edition*. Springer. ISBN 3-540-18199-7.

K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. Garcia de la Banda, and M. Wallace. 2008. The design of the Zinc modelling language. *Constraints*, 13(3): 229–267.

V. S. Mellarkod, M. Gelfond, and Y. Zhang. 2008. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53(1-4): 251–287. ISSN 1012-2443. DOI: http://dx.doi.org/10.1007/s10472-009-9116-y.

MinizincChallenge2012. Minizinc challenge 2012. http://www.minizinc.org/challenge2012/results2012.html.

D. G. Mitchell and E. Ternovska. 2005. A framework for representing and solving NP search problems. In M. M. Veloso and S. Kambhampati, eds., *AAAI*, pp. 430–435. AAAI Press / The MIT Press. ISBN 1-57735-236-X. http://www.aaai.org/Library/AAAI/2005/aaai05-068.php.

N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. 2007. Minizinc: Towards a standard CP modelling language. In C. Bessiere, ed., *CP'07*, volume 4741 of *LNCS*, pp. 529–543. Springer.

M. Ostrowski and T. Schaub. 2012. ASP modulo CSP: The clingcon system. *TPLP*, 12(4–5): 485–503.

P. Pialorsi and M. Russo. 2007. *Introducing Microsoft®Linq*, first. Microsoft Press, Redmond, WA, USA. ISBN 9780735623910.

B. C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA. ISBN 0-262-16209-1.

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

A. Van Gelder. 1993. The alternating fixpoint of logic programs with negation. *J. Comput. Syst. Sci.*, 47(1): 185–221.

A. Van Gelder, K. A. Ross, and J. S. Schlipf. 1991. The well-founded semantics for general logic programs. *J. ACM*, 38(3): 620–650. http://dx.doi.org/10.1145/116825.116838. DOI: 10.1145/116825.116838.

P. Van Hertum, I. Dasseville, G. Janssens, and M. Denecker. 2016. The KB paradigm and its application to interactive configuration. In M. Gavanelli and J. H. Reppy, eds., *Practical Aspects of Declarative Languages - 18th International Symposium, PADL 2016, St. Petersburg, FL, USA, January 18-19, 2016. Proceedings*, volume 9585 of *Lecture Notes in Computer Science*, pp. 13–29. Springer. ISBN 978-3-319-28227-5. http://dx.doi.org/10.1007/978-3-319-28228-2_2. DOI: 10.1007/978-3-319-28228-2_2.

P. Van Hertum, I. Dasseville, G. Janssens, and M. Denecker. 2017. The KB paradigm and its application to interactive configuration. *TPLP*, 17(1): 91–117. http://dx.doi.org/10.1017/S1471068416000156. DOI: 10.1017/S1471068416000156.

D. Warren, 2014. One semantics for logic programming. https://www.linkedin.com/pulse/one-semantics-logic-programming-david-warren.

C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. 2009. SPASS version 3.5. In R. A. Schmidt, ed., *CADE*, volume 5663 of *LNCS*, pp. 140–145. Springer. ISBN 978-3-642-02958-5. http://dx.doi.org/10.1007/978-3-642-02959-2_10.

J. Wittocx. May 2010. *Finite Domain and Symbolic Inference Methods for Extensions of First-Order Logic*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium.

J. Wittocx, M. Mariën, and M. Denecker. 2010. Grounding FO and FO(ID) with bounds. *J. Artif. Intell. Res. (JAIR)*, 38: 223–269.

J. Wittocx, M. Denecker, and M. Bruynooghe. Aug. 2013. Constraint propagation for first-order logic and inductive definitions. *ACM Trans. Comput. Logic*, 14(3): 17:1–17:45. ISSN 1529-3785. http://doi.acm.org/10.1145/2499937.2499938.