UPMAIL Technical Report No. 106
5 May, 1995
ISSN 1100–0686

Meta-programming with Theory Systems

Jonas Barklund
Katrin Boberg
Pierangelo Dell'Acqua
Margus Veanes

Uppsala University
Computing Science Department

Box 311, S-751 05 Uppsala, Sweden
Phone: +48−18−18 25 00
Fax: +46−18−51 19 25

Abstract

A theory system is a collection of interdependent theories, some if which stand in a meta/object relationship, forming an arbitrary number of meta-levels. The main thesis of this chapter is that theory systems constitute a suitable formalism for constructing advanced applications in reasoning and software engineering. The Alloy language for defining theory systems is introduced, its syntax is defined and a collection of inference rules is presented. A number of problems suitable for theory systems are discussed, with program examples given in Alloy. Some current implementation issues and future extensions are discussed.

This paper appears as a chapter in *Meta-logics and Logic Programming*, edited by K. Apt and F. Turini, and published by MIT Press in 1995.

1 Outline

A conventional logic program can be seen as the nonlogical axioms of a single theory. This chapter presents a thesis that we obtain a more powerful tool for applications in artificial intelligence and software engineering if we consider systems of theories, where pairs of theories may stand in an object/meta relationship, rather than single theories.

We proceed in 7 steps:

1. Arguing that multi-level programming should be a powerful tool for many advanced applications, in particular artificial intelligence and software engineering (Sect. 2).
2. Introducing theory systems as an approach to multi-level programming (Sect. 3).
3. Defining the formal syntax and one possible inference system of a language, *Alloy*, in which theory systems can be programmed (Sect. 4).
4. Defining the models of Alloy programs (Sect. 5).
5. Presenting examples of problem solving using theory systems expressed in Alloy (Sect. 6).
6. Discussing self-reference and how to program it in Alloy (Sect. 7).
7. Proposing some future extensions, supporting technologies and some current implementation issues (Sects. 8–9).

We end with some notes and conclusions.

For a general introduction to meta-programming in logic programming, the reader is referred to the overviews by Barklund [3] and Hill & Gallagher [19].

2 Artificial intelligence and software engineering

The studies of artificial intelligence in general and expert systems in particular make it clear that truly useful problem solvers must be constructed in a quite different way than has been tried in the past. Among the problems with current approaches are:

1. Lack of robustness with respect to domains.
2. Low adaptability of problem solving methods.
3. Failure to capture "common sense" reasoning.

These problems are indeed very difficult but we believe that the marginal success so far is largely because the attempts at addressing them have been carried out mostly using single-level architectures (cf. Sterling [31]). By single-level architectures we mean systems without provisions for reasoning about any part of their own beliefs or procedures and for adapting themselves according to these observations. The three problems mentioned above could be approached as follows:

1. Given a program that solves problems in some domain, the system might transform this program to adapt it to another domain. Also, given a program that represents a piece of knowledge, together with some suitably represented new knowledge, the system might create a new program that incorporates both the knowledge present in the old program and the new knowledge, after resolving any discrepancies between them.

2. Given a subprogram that carries out a particular form of reasoning, the system might transform it to a similar program that carries out a somewhat different form of reasoning, better adapted to some circumstances.

3. This is the most difficult problem of these three. McCarthy defined a program having common sense as one that "automatically deduces for itself a sufficiently wide class of immediate consequences of anything it is told and what it already knows" [27]. The heuristics for exploring the interesting consequences of new information or finding the information necessary for solving a problem are naturally expressed as meta-knowledge. These heuristics might need to be revised over time, as they turn out to be more or less successful. This can be seen as a metameta-level problem, indicating that one should not be restricted to only two levels.

All three of these potential solutions involve writing programs that are capable (i) of observing parts of other programs, (ii) of examining those programs' conclusions and perhaps also the reasoning behind these conclusions, and (iii) of creating new programs, presumably starting from existing programs.

The reader should note that the preceding sentence could just as well have been a statement about advanced software engineering; the same basic operations seem to be useful in both application areas. Our thesis is that a useful methodology for building correct software is one where a program is constructed "implicitly" by writing a meta-program that takes a number of "standard programs", transforming and combining them to produce a program that performs the desired task. The "standard programs" would

be of various kinds, some of them simple program pieces that perform various kinds of recursion, for example, but some of them might be sophisticated and complex programs that carry out a computation for the same domain as the program to be produced.

The meta-programs may in some cases be very simple, merely composing and transforming the given programs in certain ways. However, if the produced programs must satisfy particular criteria, for example, real-time constraints, then the meta-programs may have to do a much more detailed analysis or perhaps even run the generated programs as a step in their construction.

The advantage with the outlined approach is that if the standard programs are completely understood, the produced programs will be as well. Moreover, all future modifications to the produced programs are done by changing the program that generated them, which is likely to lead to fewer mistakes than manual work. This programming paradigm could truly be called "high-level programming".

Although the main body of work on artificial intelligence, reasoning and expert systems has been spent on single-level formalisms, we are certainly not alone in observing that a multilevel formalism should provide a better tool for attacking the fundamental problems. We mention some related formalisms at the end of this chapter; further references can be found in the remainder of this book.

3 Logic Programming with multiple theories

Formally, a *theory* is a set of sentences in some language, including the logical axioms of the language, that is closed under the inference rules of the language. Once the language is fixed, any set of sentences in that language defines a theory, obtained by adding the logical axioms of the language and closing it under inference. In logic programming, the language might be that of definite clauses with SLD-resolution and the logical axioms those concerning (Herbrand) equality. A program is then a set of definite clauses defining a single theory.

In applications that involve reasoning it is often appropriate to compute with more than one theory. For example, we could write a program that simulates the reasoning of a collection of agents, representing the beliefs of each agent as a theory (if we employ the "sentential" view of beliefs, perhaps first used explicitly by McCarthy [28]). If the language prevents us from having more than one theory in our program, then these "internal" theories have to be represented in some other way, perhaps as data structures with the programmer writing an ad hoc interpreter to simulate inference. There is a large class of applications in "reasoning" and software engineering, perhaps also in other areas, that are naturally written using multiple theories; therefore multiple theories ought to be supported directly in the language.

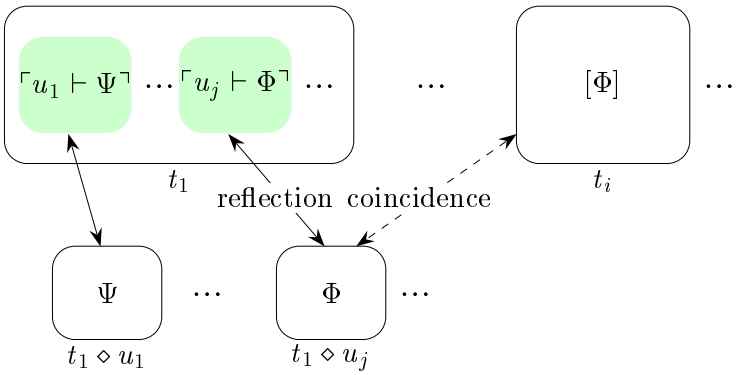

Figure 1: A generic theory system.

Theory systems constitute a useful formalism for writing these kinds of programs, because the theories in a theory system are suitable for representing reasoning agents or parts of them, programs to be manipulated, programs that manipulate them, etc. The meta/object relationship between theories provides the inspection and control facilities needed in both kinds of applications.

3.1 Theory systems

We propose now a simple structure for theory systems that appears to be adequate for our purposes. A *theory system* is a mapping from (ground) theory terms to theories. Any theory τ contains theorems about theories named $\tau \diamond \cdots$ ('$\diamond$' is a distinguished function symbol that we write using infix notation). In fact, the restriction of a theory system to theory terms of the form $\tau \diamond \cdots$, for some τ, is a theory system in itself. Such a theory system can be thought of being defined by τ.

It is convenient to say that a theory t_1 is a *meta-theory* of any theory identified as $t_1 \diamond t_2 \diamond \cdots \diamond t_k$, where $k > 1$. Conversely we say that those theories are *object theories* with respect to t_1.

We use the symbol '$\vdash$' for relating theory terms and sentences. A *theoremhood statement* $t_1 \vdash \ulcorner u_1 \vdash \Psi \urcorner$ says that $\ulcorner u_1 \vdash \Psi \urcorner$ is a theorem of t_1. We mentioned that theories may contain theorems about other theories, and $\ulcorner u_1 \vdash \Psi \urcorner$ in t_1 expresses that Ψ is a theorem in the theory $t_1 \diamond u_1$ (cf. Fig. 1). Note that a subset of the theorems of t_1, namely those on the form $\ulcorner u_1 \vdash \cdots \urcorner$ (the left shaded area in the figure), have a one-to-one correspondence with the theorems of $t_1 \diamond u_1$, and similarly for another subset of t_1 (the right shaded area) and $t_1 \diamond u_j$.

The other kind of statement that we use for defining theory systems is called a *coincidence statement*. If the program defining the theory system in Fig. 1 contains a coincidence statement $t_1 \diamond u_j \equiv t_i$, then the theories $t_1 \diamond u_j$

and t_i have exactly the same theorems. (The relation denoted by '$\equiv$' is an equivalence relation, i.e., it is reflexive, symmetric and transitive.) More importantly, that statement ensures a one-to-one correspondence between a subset of t_1 (the right shaded area) and t_i. In absence of such a coincidence statement, there is no connection whatsoever between theories, unless one is a meta-theory of the other. In particular, proving $\ulcorner t_i \vdash \Phi \urcorner$ in t_1 in order to determine in t_1 whether Φ is a theorem of t_i requires that $t_1 \diamond t_i \equiv t_i$.

3.2 Representation

We will assume that all theories use the same definite clause language but that the set of terms of this language is rich enough that for any variable, function or predicate symbol σ, there is some unique constant σ' which *represents*, or *names*, σ. Similarly, for each well-formed expression α, there must be some unique ground term α' that represents α.

Our final requirement on the definite clause language is that for any theoremhood statement and coincidence statement there is some unique ground atom representing it.

We can now define precisely the relationship between a meta-theory and an object theory. Consider a theory system and a pair of theories identified by some theory terms τ_1 and $\tau_1 \diamond \tau_2$; the first is thus a meta-theory of the second. Our *theoremhood reflection principle* states that

$$\tau_1 \vdash \ulcorner \tau_2 \vdash \kappa \urcorner \Leftrightarrow \tau_1 \diamond \tau_2 \vdash \kappa$$

and can be seen as a correctness statement for interaction between a meta-theory and an object theory.

Our *coincidence reflection principle* states that

$$\tau_1 \vdash \ulcorner \tau_2 \equiv \tau_3 \urcorner \Leftrightarrow \tau_1 \diamond \tau_2 \equiv \tau_1 \diamond \tau_3$$

and can be seen as a correctness statement for coincidence of internal theories.

Both these principles are valid for every theory system.

The traditional *local reflection principle* for a single theory T in mathematical logic [30] reads

$$Pr_T(\ulcorner \phi \urcorner) \Rightarrow \phi$$

and states the correspondence between a provability statement and what is to be proved, namely that if the provability predicate holds for an encoding of a formula ϕ, then ϕ holds as well. We call our statements reflection principles by analogy, as they state correspondences between names of statements and what these statements are about.

These implications and equivalences should not be confused with the inference rules sometimes referred to as "reflection principles" but for which

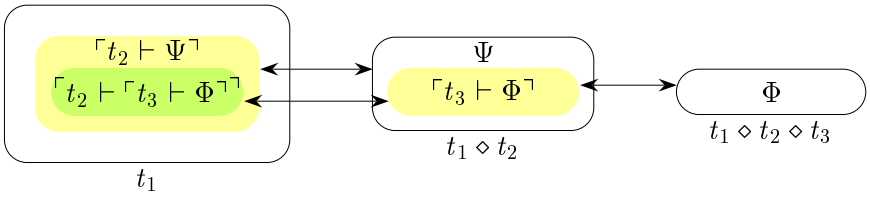

Figure 2: Three meta-levels of theories.

a better name is "reflection rules" or "linking rules" (cf. the discussions by Giunchiglia, Serafini & Simpson [18] and Costantini, Dell'Acqua & Lanzarone [15]). However, in Sect. 4.3 we will present two reflection rules corresponding to the two implications of the theoremhood reflection principle.

Fig. 2 depicts part of a theory system in which a theory t contains a theorem $\ulcorner u \vdash \ulcorner v \vdash \Phi \urcorner \urcorner$. The reflection principle requires that the theory $t \diamond u$ contains the theorem $\ulcorner v \vdash \Phi \urcorner$ and thus that $t \diamond u \diamond v$ contains Φ. The figure thus illustrates that theory systems may be arbitrarily deep and that the theoremhood reflection principle applies at any level.

Finally, the following is an example of a program defining a simple theory system.

$$Tim \vdash \ulcorner \dot{x} \vdash Tasty(\dot{x}) \urcorner \leftarrow Cannibal(y) \wedge x \; Names \; y \tag{1}$$

$$Tim \vdash Cannibal(Tom) \tag{2}$$

$$Tim \diamond Tom \equiv Tom \tag{3}$$

The theoremhood statements 1 and 2 specify two axioms of the theory *Tim*. According to the theoremhood reflection principle, statement 1 also says something about theories named $Tim \diamond \cdots$. One such theory is $Tim \diamond Tom$, which coincides with *Tom*, according to statement 3. When the theories are thought of as representing the beliefs of agents, we can read the statements as saying "Tim believes that all cannibals find themselves tasty", "Tim believes that Tom is a cannibal" and "Tim's view of Tom's beliefs is correct", respectively. From this reading we can deduce that Tom finds himself tasty, and in Sect. 4.3 we will show how to derive this conclusion using an inference system.

4 Syntax

We will now define the syntax, inference rules and informal semantics of Alloy, a language for computing with theory systems. What we define in this section can be seen as the "core" syntax of Alloy: the language of definite clauses extended with name terms, name atoms, theoremhood statements and coincidence statements.

The language, at this stage, does not contain negation, except that denials are introduced as part of proving goals (as usual in SLD-resolution).

In Sect. 6.2 we use negation in some examples, which are therefore not meaningful until Alloy is extended with negation.

4.1 Formal syntax

The Alloy language has two components: the system component for defining theory systems and the theory component for defining individual theories.

Alphabet. Besides punctuation symbols, the part of the alphabet that is common to both components of the language consists of a class of *variables* and, for each $n \geq 0$, a class of *function symbols* and a class of *predicate symbols* of *arity* n. Collectively, function and predicate symbols are referred to as *functors*. As usual, function and predicate symbols of arity 0 are referred to as *constants* and *propositional constants*, respectively. The class of predicate symbols include the binary symbols '$=$' and '*Name*', two binary symbols that will be denoted by '$\vdash'$' and '$\equiv'$', and the propositional constants '*True*' and '*False*'. The function symbols include the binary symbol '$\diamond$'.

The alphabet has also a collection of *connectives* '$\leftarrow$', '$\wedge$' and '?', and *naming symbols* '$\ulcorner$', '$\urcorner$', '$\cdot$' ("dot") and '$|$'. A dot is used in combination with variables only. If x is a variable, then $\dot{x}$ is called a variable *with a dot*, $\dot{\dot{x}}$ a variable *with two dots*, etc., in general a variable with one or more dots is called a *dotted* variable.

In addition, the system component of the language has the binary operators '$\vdash$' and '$\equiv$'.

We will use letters P, Q and R to stand for predicate symbols, F and G to stand for function symbols, x, y and z to stand for variables.[1]

Theory component. In the following we define the expressions of the theory component. We will do so by a simultaneous inductive definition of terms, atoms, queries and sentences as separate subclasses and refer to them collectively as *theory expressions*.

In the definition of terms and atoms we will make use of the notion of an expression being a schema of another. Intuitively, E is a schema of an expression e whenever some (or none) subexpression occurrences of e have been replaced by dotted variables ("holes") in E. In general, we say that E is a *k-level schema* of e if one of the following conditions holds.

1. $e = E$,
2. e is not a connective and E is a variable with k dots,
3. $e = e_0(e_1, \ldots, e_n)$ and
   - $E = E_0(E_1, \ldots, E_n)$, or

[1] The letters may also be subscripted.

- $E = E_0(E_1, \ldots, E_j|X)$ for some j, $0 \leq j < n$,

where each E_i is a k-level schema of e_i, $0 \leq i \leq n$, and X a variable with k dots.

4. $e = \ulcorner d \urcorner$ and $E = \ulcorner D \urcorner$, where D is a $(k+1)$-level schema of d.

By simply *schema* we mean a 1-level schema. It is an immediate consequence of the definition that if E is a schema of e and a subexpression of e has been replaced by a variable with k dots in E, then this dotted variable occurs nested within $k-1$ pairs of '$\ulcorner$' and '$\urcorner$'. For example, if F is a binary functor, then $\dot{x}(y, \ulcorner G(\dot{\dot{z}}) \urcorner)$ is a schema of $F(y, \ulcorner G(\dot{z}) \urcorner)$.

Terms. The class of *terms* is the least class satisfying the following conditions.

1. Each variable and constant is a term.
2. If F is a function symbol of arity n and $t_1, t_2, \ldots, t_n$ are terms, then $F(t_1, t_2, \ldots, t_n)$ is a term.
3. If X is a schema of a functor or a theory expression, then $\ulcorner X \urcorner$ is a term, called a *name term*.

Letters t and u will be used for terms.

Atoms. The class of *atoms* is the least class satisfying the following conditions.

1. If P is a predicate symbol of arity n and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is an atom, called a *predication*.
2. If T is a schema of a term t and S a schema of a sentence s, then $\vdash'(\ulcorner T \urcorner, \ulcorner S \urcorner)$ is an atom, called a *name atom*.
3. If T is a schema of a term t and U a schema of a term u, then $\equiv'(\ulcorner T \urcorner, \ulcorner U \urcorner)$ is an atom, called a *name atom*.

Letters A and B will be used for atoms. We will use the shorthand $\ulcorner T \vdash S \urcorner$ for $\vdash'(\ulcorner T \urcorner, \ulcorner S \urcorner)$ and $\ulcorner T \equiv U \urcorner$ for $\equiv'(\ulcorner T \urcorner, \ulcorner U \urcorner)$.[2]

It follows easily from the definitions of terms and atoms that a variable with n dots is embedded within k, $k \geq n$, nested levels of naming. For a variable with n dots, the lowest dot makes a "hole" in the innermost pair of '$\ulcorner$' and '$\urcorner$', the next dot in the next pair and so on. If $k = n$, then the

[2]We could let $\vdash'$ be *Demo*, in which case $\ulcorner T \vdash S \urcorner$ would be shorthand for the familiar $Demo(\ulcorner T \urcorner, \ulcorner S \urcorner)$.

variable is called free in the corresponding term or atom. For example z is the only free variable of the name term

$$\ulcorner\ulcorner F(x, \dot{y}, \dot{\dot{z}})\urcorner\urcorner.$$

A name term or name atom is said to be *proper* if it is ground. Consider for example

$$\ulcorner F(\dot{x_1}) \vdash \dot{x_2}(\dot{x_1}, y, A) \leftarrow \dot{z}\urcorner.$$

This name atom is not proper because it contains four free variable occurrences of three different variables. Only for proper names can we tell which expression they name.

Queries. The class of *queries* is the least class satisfying the following conditions.

1. An atom is a query; *True* is called the *empty query*.
2. If C and D are queries, then $C \wedge D$ is a query.

Letters C and D will be used for queries.

Sentences. The class of *sentences* is the least class satisfying the following conditions.

1. If A is an atom and C is a query, then $A \leftarrow C$ is a sentence, called a *program clause*.
2. If C is a query, then $C?$ is a sentence, called a *goal*.
3. If C is a query, then $\leftarrow C$ (shorthand for $\mathit{False} \leftarrow C$) is a sentence, called a *denial*.

The variables of a program clause and a denial are universally quantified. A goal, on the other hand, is the negation of a denial and thus existentially quantified:

$$\neg(\leftarrow C) \Leftrightarrow \neg(\forall(\mathit{False} \leftarrow C)) \Leftrightarrow \neg(\mathit{False} \leftarrow \exists C) \Leftrightarrow \exists C \Leftrightarrow C?.$$

System component. The language of the system component has two kinds of expressions: theoremhood statements and coincidence statements.

- If t is a term (called a *theory term* in this context) and s is a sentence, then $t \vdash s$ is a *theoremhood statement*.
- If t_1 and t_2 are (theory) terms, then $t_1 \equiv t_2$ is a *coincidence statement*.

Collectively they are referred to as *system expressions*.

4.2 Normalized language

In order to be able to handle terms conventionally, we want each term to have a *normal form*, where the naming symbols '$\ulcorner$', '$\urcorner$', '$\cdot$' and '$|$' have been eliminated. We call the elimination process *normalization* and the result a *normalized term*. In this context $\ulcorner - \urcorner$ is a function mapping expressions to expressions; $\ulcorner - \urcorner$ is required to be *compositional* in order to enhance the expressive power of the language. This means that if e is a compound expression $e_0(e_1, \ldots, e_n)$, then $\ulcorner e \urcorner$ can be expressed as a composition of all $\ulcorner e_i \urcorner$, $0 \leq i \leq n$. In addition, $\ulcorner \dot{v} \urcorner = v$.

Clearly there exist several different normalizations. Probably the most general approach is to have a binary function symbol '$\circ$', denoting a composition function that produces the name of a compound expression from the name of a functor or a connective and a list of names of expressions. Using this approach, the notion of lists is needed; this can be accomplished by using a binary function symbol '$\bullet$' and a constant 'Λ' to represent the empty list. (We will use the less cumbersome notation $[e_1, e_2, \ldots, e_n|x]$ for $\bullet(e_1, \bullet(e_2, \cdots \bullet(e_n, x) \cdots))$.)

The alphabet is assumed to have a unique name e' for each symbol e, in such a way that the mapping $e \mapsto e'$ is injective.[3] If these names are all terms, then the normalization can be described by the following transformations.

$$\begin{array}{rcl}
\ulcorner \dot{e} \urcorner & \longrightarrow & e \\
\ulcorner e \urcorner & \longrightarrow & e' \quad \text{if } e \text{ is a symbol} \\
\ulcorner e_0(e_1, \ldots, e_n|\dot{x}) \urcorner & \longrightarrow & \circ(\ulcorner e_0 \urcorner, [\ulcorner e_1 \urcorner, \ldots, \ulcorner e_n \urcorner|x]) \\
\ulcorner e_0(e_1, \ldots, e_n) \urcorner & \longrightarrow & \circ(\ulcorner e_0 \urcorner, [\ulcorner e_1 \urcorner, \ldots, \ulcorner e_n \urcorner])
\end{array}$$

We can however take advantage of the restriction that we imposed on the definition of schemas, namely disallowing holes for connectives, and make the following modifications to the above transformations. For each connective c, the alphabet has a corresponding function symbol c' of the same arity as c. If e_0 in the last case above is for example '$\wedge$', then

$$\ulcorner e_1 \wedge e_2 \urcorner \quad \longrightarrow \quad \wedge'(\ulcorner e_1 \urcorner, \ulcorner e_2 \urcorner).$$

We get similar transformations for the other connectives. For example, using this normalization, the normal form of $\ulcorner \ulcorner F(x, \dot{y}, \dot{\dot{z}}) \urcorner \urcorner$ is obtained as follows.

$$\begin{array}{rcl}
\ulcorner \ulcorner F(x, \dot{y}, \dot{\dot{z}}) \urcorner \urcorner & \stackrel{*}{\longrightarrow} & \ulcorner \circ(F', [x', y, \dot{z}]) \urcorner \\
& \stackrel{*}{\longrightarrow} & \circ(\circ', [F'', \ulcorner [x', y, \dot{z}] \urcorner]) \\
& \stackrel{*}{\longrightarrow} & \circ(\circ', [F'', \circ(\bullet', [x'', \ulcorner [y, \dot{z}] \urcorner])])
\end{array}$$

[3] If e' itself is a symbol it has a name e'', etc.

$$\begin{array}{rl} \xrightarrow{*} & \circ(\circ', [F'', \circ(\bullet', [x'', \circ(\bullet', [y', \ulcorner[\dot{z}]\urcorner])])]) \\ \xrightarrow{*} & \circ(\circ', [F'', \circ(\bullet', [x'', \circ(\bullet', [y', \circ(\bullet', [z, \Lambda'])])])]) \end{array}$$

Unnecessary naming of '$\circ$', '$\bullet$' and 'Λ' can be avoided by defining the transformation so that '$\circ$', '$\bullet$' and 'Λ' become "transparent" with respect to naming, i.e., $\ulcorner\Lambda\urcorner \longrightarrow \Lambda$, $\ulcorner\circ(e_1, e_2)\urcorner \longrightarrow \circ(\ulcorner e_1\urcorner, \ulcorner e_2\urcorner)$ and $\ulcorner\bullet(e_1, e_2)\urcorner \longrightarrow \bullet(\ulcorner e_1\urcorner, \ulcorner e_2\urcorner)$. Note that this does not violate the injectivity of the naming function. Assuming this modification then, for example,

$$\ulcorner\ulcorner F(x, \dot{y}, \dot{z})\urcorner\urcorner \xrightarrow{*} \circ(F'', [x'', y', z]).$$

4.3 Inference system

Equality, naming and unification. Before normalizing the language of a program, we extend each of its theories with every axiom on the form $\ulcorner t\urcorner$ *Names* t, where t is a term.

After normalizing, as described in Sect. 4.2, the usual Herbrand equality theory, as axiomatized by Clark [14], can be used. (However, computation of the naming relation ought to be integrated with unification in order to delay computation of names of nonground terms.)

As all correct normalizations will behave in the same way, with respect to equality of the normalized expressions, it would alternatively be conceivable to extend Herbrand equality to name expressions without normalization.

Inference rules. The inference system that we are to explain here is by no means the only possible inference system for Alloy, in fact, it is not even complete. We choose this inference system for presentation because it is simple and because it is complete for propositional programs. For an actual implementation we are presently developing a more goal-oriented inference system, outlined in Sect. 9.

The main purpose of the inference system is to be able to prove statements of the form $\tau \vdash C?$, i.e., that a goal $C?$ is a theorem of some theory τ. This can either be accomplished by a refutation, i.e., by assuming $\tau \vdash \leftarrow C$ and proving $\tau \vdash \leftarrow$ (inconsistency in τ), or by a proof that may include refutations as subproofs. A successful refutation of a denial $\tau \vdash \leftarrow C$ is always ended by cancelling $\tau \vdash \leftarrow C$ and concluding $\tau \vdash C?$ (through the application of the RR rule described below).

We shall present seven inference rules. The first rule is ordinary SLD-resolution within a theory. Let $\triangle C$ denote the atom selected from a query C and $\triangledown C$ the rest of the query. It is assumed that the predicate symbol of the selected atom is not *Names*.

$$\text{RS}\ \frac{\tau \vdash \leftarrow C \quad \tau \vdash A \leftarrow D}{\tau \vdash (\leftarrow \triangledown C \wedge D)\theta} \quad \theta = mgu(A, \triangle C)$$

The second rule is a "Relativized RAA" rule, allowing us to make subproofs that are refutations.

$$\dfrac{\begin{array}{c}[\tau \vdash \leftarrow C]^i \\ \vdots \\ \tau \vdash \leftarrow\end{array}}{\tau \vdash C?}\ \text{RR cancel } i$$

The third and fourth rules are the reflection rules, justifiable from the theoremhood reflection principle (Sect. 3.2). They make use of the meta/object relationship between a pair of theories in both directions: If a theory τ_M reasons that its internal theory τ_O contains some sentence κ, then $\tau_M \diamond \tau_O$ indeed contains κ, and vice versa.

$$\text{TD}\ \dfrac{\tau_M \vdash \ulcorner \tau_O \vdash \kappa \urcorner}{\tau_M \diamond \tau_O \vdash \kappa} \qquad \dfrac{\tau_M \diamond \tau_O \vdash \kappa}{\tau_M \vdash \ulcorner \tau_O \vdash \kappa \urcorner}\ \text{TU}$$

The fifth and sixth rules are similar to the third and fourth, but are instead justifiable from the coincidence reflection principle. They express that if a theory has as a theorem stating that two of its internal theories coincide, then we may infer that these theories do coincide, and vice versa.

$$\text{CD}\ \dfrac{\tau \vdash \ulcorner \tau_1 \equiv \tau_2 \urcorner}{\tau \diamond \tau_1 \equiv \tau \diamond \tau_2} \qquad \dfrac{\tau \diamond \tau_1 \equiv \tau \diamond \tau_2}{\tau \vdash \ulcorner \tau_1 \equiv \tau_2 \urcorner}\ \text{CU}$$

The seventh rule uses a coincidence between two theories to transfer a theorem of one of them to the other.

$$\dfrac{\tau_1 \equiv \tau_2 \quad \tau_1 \vdash \kappa}{\tau_2 \vdash \kappa}\ \text{CE}$$

From these inference rules one could derive others, for example, an indirect SLD-resolution inference.

$$\dfrac{\tau_M \vdash \ulcorner \tau_O \vdash \leftarrow C \urcorner \quad \tau_M \vdash \ulcorner \tau_O \vdash A \leftarrow D \urcorner}{\tau_M \vdash \ulcorner \tau_O \vdash (\leftarrow \bigtriangledown C \wedge D)\theta \urcorner} \quad \theta = mgu(A, \bigtriangleup C)$$

This derived inference rule can be justified:

$$\text{TD}\ \dfrac{\dfrac{\dfrac{\tau_M \vdash \ulcorner \tau_O \vdash \leftarrow C \urcorner}{\tau_M \diamond \tau_O \vdash \leftarrow C} \quad \dfrac{\tau_M \vdash \ulcorner \tau_O \vdash A \leftarrow D \urcorner}{\tau_M \diamond \tau_O \vdash A \leftarrow D}\ \text{TD}}{\tau_M \diamond \tau_O \vdash (\leftarrow \bigtriangledown C \wedge D)\theta}\ \text{RS}}{\tau_M \vdash \ulcorner \tau_O \vdash (\leftarrow \bigtriangledown C \wedge D)\theta \urcorner}\ \text{TU}$$

Another useful derived rule is for indirect reasoning with coinciding theories,

$$\dfrac{\tau \vdash \ulcorner \tau_1 \equiv \tau_2 \urcorner \quad \tau \vdash \ulcorner \tau_1 \vdash \kappa \urcorner}{\tau \vdash \ulcorner \tau_2 \vdash \kappa \urcorner},$$

justified as follows.

$$\dfrac{\dfrac{\dfrac{\tau \vdash \ulcorner \tau_1 \equiv \tau_2 \urcorner}{\tau \diamond \tau_1 \equiv \tau \diamond \tau_2}\,\text{CD} \quad \dfrac{\tau \vdash \ulcorner \tau_1 \vdash \kappa \urcorner}{\tau \diamond \tau_1 \vdash \kappa}\,\text{TD}}{\tau \diamond \tau_2 \vdash \kappa}\,\text{CE}}{\tau \vdash \ulcorner \tau_2 \vdash \kappa \urcorner}\,\text{TU}$$

As an example, consider again the cannibal example of Sect. 3.1. Here is how to prove the statement $Tom \vdash Tasty(Tom)?$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(1) \quad [Tim \vdash \leftarrow \ulcorner Tom \vdash Tasty(Tom) \urcorner]^1}{(2) \quad Tim \vdash \leftarrow Cannibal(y) \wedge \ulcorner Tom \urcorner\ Names\ y}\,\text{RS}}{Tim \vdash \leftarrow \ulcorner Tom \urcorner\ Names\ Tom}\,\text{RS}}{Tim \vdash \leftarrow True}\,\text{NM}}{Tim \vdash \ulcorner Tom \vdash Tasty(Tom) \urcorner?}\,\text{RR cancel 1}}{(3) \quad Tim \diamond Tom \vdash Tasty(Tom)?}\,\text{TD}}{Tom \vdash Tasty(Tom)?}\,\text{CE}$$

We mentioned above that this inference system is incomplete. What must be done in order to increase the number of provable statements is taking care of proofs that involve improper names. See Sect. 9 for a further discussion of how this can be done.

5 Semantics

Let $\mathcal{I}$ be the set of theory terms and $\mathcal{M} = \{\mathfrak{M}_\tau\}_{\tau \in \mathcal{I}}$ a family of (arbitrary first order[4]) structures for the language of theory expressions under a given normalization. The elements of $\mathcal{M}$ are called *theory structures*. A *system structure* is a pair $\langle \mathcal{M}, \equiv \rangle$, where $\equiv$ is an elementary equivalence relation on $\mathcal{M}$. (Two first order structures are said to be elementarily equivalent whenever they have the same set of logical consequences.)

Let P be an Alloy program, i.e., a set of system expressions, and let $\langle \mathcal{M}, \equiv \rangle$ be a system structure. If the theory structures are Herbrand interpretations, we can assume without loss of generality that P is ground; P could then be a Herbrand instantiation (possibly infinite) of an underlying nonground program. We say that $\langle \mathcal{M}, \equiv \rangle$ is a *model* of P if the following hold:

$$\tau \vdash \varphi \in P \quad \Rightarrow \quad \mathfrak{M}_\tau \models \varphi; \tag{4}$$

$$\tau_1 \equiv \tau_2 \in P \quad \Rightarrow \quad \mathfrak{M}_{\tau_1} \equiv \mathfrak{M}_{\tau_2}; \tag{5}$$

$$\mathfrak{M}_{\tau_1} \models \vdash'(\ulcorner \tau_2 \urcorner, \ulcorner \varphi \urcorner) \quad \Leftrightarrow \quad \mathfrak{M}_{\tau_1 \diamond \tau_2} \models \varphi; \tag{6}$$

[4]Here we need not restrict ourselves to Herbrand interpretations only.

$$\mathfrak{M}_{\tau_1} \models \equiv'(\ulcorner\tau_2\urcorner, \ulcorner\tau_3\urcorner) \quad \Leftrightarrow \quad \mathfrak{M}_{\tau_1 \diamond \tau_2} \equiv \mathfrak{M}_{\tau_1 \diamond \tau_3}; \tag{7}$$

$$\mathfrak{M}_{\tau_1 \diamond (\tau_2 \diamond \tau_3)} \equiv \mathfrak{M}_{(\tau_1 \diamond \tau_2) \diamond \tau_3} \tag{8}$$

$$t \text{ is a ground term} \quad \Rightarrow \quad \mathfrak{M}_{\tau} \models \ulcorner t \urcorner \; \mathit{Names}\; t. \tag{9}$$

Conceptually, the set of theoremhood statements of P is partitioned by the theory terms. Each part is identified by a theory term, the denotation of which is a model for that part (4). A coincidence statement between any two theory terms enforces the structures they denote to be elementarily equivalent (5). The theoremhood and coincidence reflection principles must be satisfied by the theory structures (6, 7). Furthermore, $\diamond$ must be associative with respect to elementary equivalence between the denoted structures (8). From formulas 6 and 8 we can easily deduce that

$$\mathfrak{M}_{\tau_1} \models \ulcorner \tau_2 \diamond \tau_3 \vdash \varphi \urcorner \Leftrightarrow \mathfrak{M}_{\tau_1} \models \ulcorner \tau_2 \vdash \ulcorner \tau_3 \vdash \varphi \urcorner \urcorner.$$

Finally, the *Names* predicate symbol must denote a naming relation (restricted to terms), i.e., one that relates any ground term with its name (9). It is also clear that the set of logical consequences of any theory structure is closed under SLD-resolution, as the set of theorems of any first order structure is complete.

Considering the special case when P is just a Horn clause program, i.e., when all the sentences of P are of the the form $\tau \vdash \varphi$ where φ is a Horn clause and τ is the only theory term, then the notion of system structure collapses to that of a first order structure. In that case conditions 5–7 are trivially satisfied. The only extra requirement, not part of a standard definition of a model of P, would be (9).

In our approach we have not altered the notion of logical consequence, as was done for example by Jiang [20], in order to handle meta-reasoning. Instead we introduce the notion of system structure, following closely the informal semantics, giving us a notion of semantics which is a modest extension of a first order semantics in the sense that the basic building blocks, theory structures, are still first order structures. A more thorough investigation of the semantics of Alloy will be the subject of a future publication.

6 Applications using theory systems

In this section we shall present a number of useful applications of meta-programming with theory systems, some of them commonly known, some of them new. We shall show how fragments of these applications can be programmed elegantly in Alloy. Our ambition is twofold. Firstly, we wish to convince the reader of the strength and versatility of meta-programming with theory systems, continuing and extending the work by Bowen & Kowalski [8], Sterling [31], Bowen [7], Brogi & Turini [12] and others. Secondly, we hope to illustrate programming in Alloy and how many problems can

be programmed in a much more straightforward and concise way than in single-level programming or single-theory meta-programming.

6.1 Reasoning Agents

Many forms of reasoning for artificial agents have been proposed, such as abductive reasoning, inductive reasoning, non-monotonic reasoning, case based reasoning, temporal reasoning and so on. A favourite approach of many philosophers and other researchers in artificial intelligence is to invent a new specialized logic for each one of these forms of reasoning. There are many problems with this approach. One is that it is not clear at all that these logics can be combined to build artificial agents capable of more than one form of reasoning. Another is that there are often no efficient implementation techniques known for these new logics.

A more sensible method is to employ a single logic, with known properties, which can be implemented; such as some subset of classical logic. However, many of the forms of reasoning mentioned above cannot be mapped straightforwardly to classical logic. (This has even been used as an argument against using logic at all for reasoning agents.)

Fortunately, there is a partial solution. If we go from using single-level logic languages to meta-logic languages for theory systems, we obtain a modest extension of classical logic in terms of semantics but we get a substantial extension in terms of reasoning capabilities, because we can express various forms of reasoning in the logic itself. This approach becomes even more sensible when one recognizes that many forms of reasoning actually contain a substantial element of meta-level reasoning. For example, default reasoning involves observing that some question cannot be decided and making a hypothesis (although it is not always recognized as such) about the answer.

In Alloy we can represent an agent's beliefs by a theory, which internally defines a system of theories. Some of these theories might represent (correctly or incorrectly) the agent's view of other agents' beliefs, ambitions and motives; cf. Fig. 3. Other theories might represent the agent's beliefs about the surroundings and about various domains. Presumably, there are also theories that encode various problem solving strategies and tactics.

This approach has several advantages.

- Modularity. An agent's mind is internally structured.
- Multiple levels. It is possible to represent beliefs and procedures at various meta-levels, e.g., theories synthesizing problem solving procedures to be used in specific domains represented by "lower" theories.
- No parapsychology. As the theory representing the beliefs of an agent is clearly separated from the theory representing another agent's beliefs about the first agent's beliefs, our formalism does not create "mind-reading" and confusion (unless explicitly programmed).

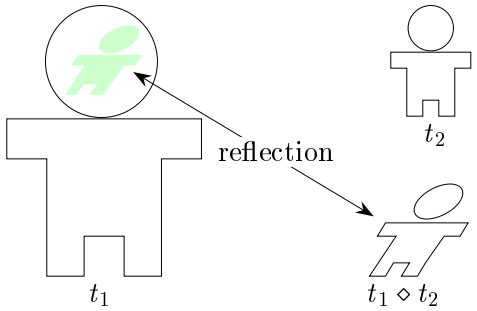

Figure 3: An agent t_1, which has a (distorted) view of the beliefs of another agent t_2, constituting a theory $t_1 \diamond t_2$.

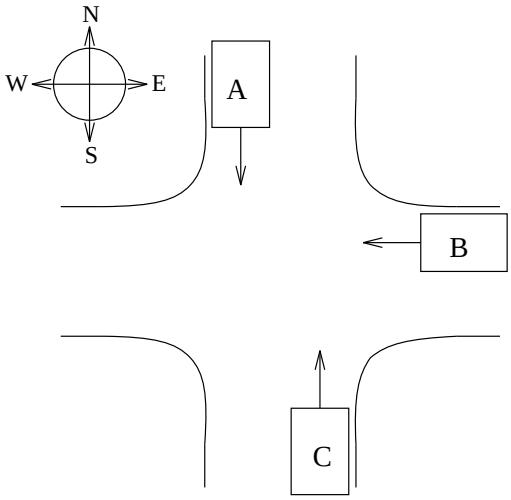

Figure 4: "Can the driver in car C, coming from south, pass the crossing?"

- Generality. Various properties of knowledge and beliefs (see the following section) can be programmed into the system but they are not automatically present.

As an example of programming multiple agents that reason about each other, consider the traffic problem illustrated in Fig. 4.

Three cars are simultaneously approaching a four-way crossing. There are no other signs or traffic lights, so the rule is that drivers should give way to cars coming on their right side. Using a simple application of this rule, we obtain that car A can pass, while cars B and C must wait, because they give way to some car on their right side. However, the driver of car C could instead reason that car B must wait, because the driver of car B will see car A on her right entering the crossing and give way to it. Hence, the

driver of car C might conclude that he can safely pass.

Our purpose here is not to argue whether it would be legal or not for the driver of car C to pass, based on the argument above[5], but to show that such multiagent reasoning can be programmed straightforwardly in Alloy.

The following statement encodes the problem of the driver of car C.

$$\mathit{Traffic} \vdash \ulcorner D(C, \mathit{South}) \vdash \mathit{Pass}([D(A, \mathit{North}), D(B, \mathit{East}), D(C, \mathit{South})])?\urcorner \tag{10}$$

The theory *Traffic* is where our reasoning about the drivers will take place. Each theory $\mathit{Traffic} \diamond D(x, y)$ represents (our view of) the beliefs of the driver of car x, coming from direction y. Each theory $\mathit{Traffic} \diamond D(x_1, y_1) \diamond D(x_2, y_2)$ represents (our view of) the beliefs that the driver of car x_1, coming from direction y_1 has about the beliefs of the driver of car x_2, coming from direction y_2 has, etc.

A theorem $\mathit{Pass}(z)$ in a theory $\cdots \diamond D(x, y)$ would mean that the driver in question would believe that she can pass a crossing in which she sees the cars listed in z. Similarly, a theorem $\mathit{Wait}(x)$ would mean that she would believe she has to stop.

The first two clauses are interesting, because they help us to encode a form of *group belief*.

$$\begin{aligned} &\mathit{Traffic} \vdash \mathit{Driver}(D(x, y), D(x, y)) \\ &\mathit{Traffic} \vdash \mathit{Driver}(D(x, y) \diamond p, d) \leftarrow \mathit{Driver}(p, d) \end{aligned}$$

Every atom on the form $\mathit{Driver}(D(x_1, y_1) \diamond \cdots \diamond D(x_n, y_n), D(x_n, y_n))$ is a theorem in *Traffic*. For example, we can derive $\mathit{Traffic} \vdash \mathit{Driver}(D(C, \mathit{South}), D(C, \mathit{South}))$ and $\mathit{Traffic} \vdash \mathit{Driver}(D(C, \mathit{South}) \diamond D(B, \mathit{East}) \diamond D(A, \mathit{North}), D(A, \mathit{North}))$. Note that each such theorem is about a theory term encoding some driver's view of some driver's view of ... some driver's beliefs, and the ultimate driver in such a chain. We can use the predicate *Driver* in *Traffic* for expressing that something should be believed by every driver and that every driver should believe that other drivers believe so, etc., arbitrarily deep.

[5]It is easy to observe that many real drivers seem to reason exactly this way.

The following three clauses define the actual reasoning.

$$\begin{aligned}
&\mathit{Traffic} \vdash \ulcorner \dot{t_1} \vdash \mathit{Pass}(c) \leftarrow \mathit{Not\text{-}in\text{-}crossing}(\dot{x_1}, c)\urcorner \leftarrow \\
&\qquad \mathit{Driver}(t,d) \wedge t_1 \ \mathit{Names}\ t \wedge x_1 \ \mathit{Names}\ x \wedge \mathit{Gives\text{-}way\text{-}to}(d,x) \\
&\mathit{Traffic} \vdash \ulcorner \dot{t_1} \vdash \mathit{Wait}(c) \leftarrow \mathit{In\text{-}crossing}(\dot{x_1}, c) \wedge \\
&\qquad\qquad x_2 \ \mathit{Names}\ x_1 \wedge c_1 \ \mathit{Names}\ c \wedge \\
&\qquad\qquad \ulcorner \dot{x_2} \vdash \mathit{Pass}(\dot{c_1})?\urcorner\urcorner \leftarrow \\
&\qquad \mathit{Driver}(t,d) \wedge t_1 \ \mathit{Names}\ t \wedge x_1 \ \mathit{Names}\ x \wedge \mathit{Gives\text{-}way\text{-}to}(d,x) \\
&\mathit{Traffic} \vdash \ulcorner \dot{t_1} \vdash \mathit{Pass}(c) \leftarrow \mathit{In\text{-}crossing}(\dot{x_1}, c) \wedge \\
&\qquad\qquad x_2 \ \mathit{Names}\ x_1 \wedge c_1 \ \mathit{Names}\ c \wedge \\
&\qquad\qquad \ulcorner \dot{x_2} \vdash \mathit{Wait}(\dot{c_1})?\urcorner\urcorner \leftarrow \\
&\qquad \mathit{Driver}(t,d) \wedge t_1 \ \mathit{Names}\ t \wedge x_1 \ \mathit{Names}\ x \wedge \mathit{Gives\text{-}way\text{-}to}(d,x)
\end{aligned}$$

The first clause says that any driver will reason: if there is no car approaching from such a direction that I must give way to it, then I may pass.

The second clause says that any driver will reason: if there is a car approaching from such a direction that I must give way to it, and I believe that driver will reason that he can pass, then I must wait.

The third clause says that any driver will reason: if there is a car approaching from such a direction that I must give way to it, but I believe that driver will reason that he must wait, then I can pass anyway.

The next four clauses of *Traffic* simply determine who must yield to whom.

$$\begin{aligned}
&\mathit{Traffic} \vdash \mathit{Gives\text{-}way\text{-}to}(D(_, \mathit{North}), D(_, \mathit{West})) \\
&\mathit{Traffic} \vdash \mathit{Gives\text{-}way\text{-}to}(D(_, \mathit{West}), D(_, \mathit{South})) \\
&\mathit{Traffic} \vdash \mathit{Gives\text{-}way\text{-}to}(D(_, \mathit{South}), D(_, \mathit{East})) \\
&\mathit{Traffic} \vdash \mathit{Gives\text{-}way\text{-}to}(D(_, \mathit{East}), D(_, \mathit{North}))
\end{aligned}$$

The predicates *In-crossing* and *Not-in-crossing* are list membership/non-membership predicates; these predicates are here part of the group belief of drivers (alternatively we could have placed them in every theory $\mathit{Traffic}\diamond\cdots$).

$$\begin{aligned}
&\mathit{Traffic} \vdash \ulcorner \dot{t_1} \vdash \mathit{In\text{-}crossing}(x, [x|_])\urcorner \leftarrow \\
&\qquad \mathit{Driver}(t,d) \wedge t_1 \ \mathit{Names}\ t \\
&\mathit{Traffic} \vdash \ulcorner \dot{t_1} \vdash \mathit{In\text{-}crossing}(x, [_|c]) \leftarrow \mathit{In\text{-}crossing}(x,c)\urcorner \leftarrow \\
&\qquad \mathit{Driver}(t,d) \wedge t_1 \ \mathit{Names}\ t \\
&\mathit{Traffic} \vdash \ulcorner \dot{t_1} \vdash \mathit{Not\text{-}in\text{-}crossing}(x, [])\urcorner \leftarrow \\
&\qquad \mathit{Driver}(t,d) \wedge t_1 \ \mathit{Names}\ t \\
&\mathit{Traffic} \vdash \ulcorner \dot{t_1} \vdash \mathit{Not\text{-}in\text{-}crossing}(x, [y|c]) \leftarrow x \neq y \wedge \mathit{Not\text{-}in\text{-}crossing}(x,c)\urcorner \leftarrow \\
&\qquad \mathit{Driver}(t,d) \wedge t_1 \ \mathit{Names}\ t
\end{aligned}$$

A full proof of the original statement 10 is rather long, but involves proving

the following statements, among others.

$$\begin{array}{l} \mathit{Traffic} \diamond D(C, \mathit{South}) \diamond D(B, \mathit{East}) \diamond D(A, \mathit{North}) \vdash \\ \quad \mathit{Pass}([D(A, \mathit{North}), D(B, \mathit{East}), D(C, \mathit{South})])? \\ \mathit{Traffic} \diamond D(C, \mathit{South}) \diamond D(B, \mathit{East}) \vdash \\ \quad \mathit{Wait}([D(A, \mathit{North}), D(B, \mathit{East}), D(C, \mathit{South})])? \\ \mathit{Traffic} \diamond D(C, \mathit{South}) \vdash \mathit{Pass}([D(A, \mathit{North}), D(B, \mathit{East}), D(C, \mathit{South})])? \\ \mathit{Traffic} \vdash \ulcorner D(C, \mathit{South}) \vdash \mathit{Pass}([D(A, \mathit{North}), D(B, \mathit{East}), D(C, \mathit{South})])? \urcorner \end{array}$$

6.2 Properties of Knowledge

Some formalisms intended for knowledge representation, reasoning and meta-reasoning (such as Konolige's modal logic of knowledge [22]) build various properties of knowledge or belief into the formalism. Five well-known properties of this kind are (using the notation of Konolige, where $bel(S)$ is the set of beliefs of an agent S, while $[S]\phi$ is the proposition that agent S believes ϕ):

Saturation (K). Reasoners are closed under inference, so $bel(S)$ is saturated.

Knowledge (T). For knowledge, beliefs must be true, so $\phi \in bel(S) \Rightarrow \phi$ is true.

Consistency (D). Reasoners are supposed to be consistent in their knowledge, so $\phi \in bel(S) \Rightarrow \neg\phi \notin bel(S)$.

Positive introspection (4). If reasoners believe something, they also believe that they believe it, so $\phi \in bel(S) \Rightarrow [S]\phi \in bel(S)$.

Negative introspection (5). If reasoners do not believe something, they also believe that they do not believe it, so $\phi \notin bel(S) \Rightarrow \neg[S]\phi \in bel(S)$.

Alloy is intended, among other things, for applications of this kind, but only the first property has been built into the language. Instead, we might express these properties as part of our meta-programs. This makes it possible to model also reasoning agents that do not have these properties, or who have quite different properties. Let us show how these properties could be represented in a suitably extended version of Alloy, one by one. We will assume that there is a theory A which defines an internal theory system, in which the beliefs of some agent is represented by a theory identified as B in A (and thus as $A \diamond B$ outside A).

Saturation (K) This property is built in, as Alloy theories are closed under inference. This means that Alloy can only represent directly agents whose beliefs are closed under inference.

Knowledge (T) This postulate can be expressed for some particular binary predicate P as a theoremhood statement

$$A \vdash P(x, y) \leftarrow \ulcorner B \vdash P(\dot{u}, \dot{v})? \urcorner \wedge u \; Names \; x \wedge v \; Names \; y.$$

If we would like to express the **T** postulate for *any* predicate symbol we should do it in a meta-theory of A.

A variant of the **T** postulate can be expressed as

$$A \vdash \ulcorner W \vdash P(\dot{u}, \dot{v}) \urcorner \leftarrow \ulcorner B \vdash P(\dot{u}, \dot{v})? \urcorner,$$

in which A has an internal theory W which contains A's view of the world. This statement says that if A believes that B believes some P atom, then that atom is also contained in A's beliefs about the world.

Consistency (D) Consistency of the reasoner B could be expressed as an integrity constraint

$$A \vdash \; \leftarrow \ulcorner B \vdash \dot{p} \urcorner \wedge \ulcorner B \vdash not \, \dot{p} \urcorner$$

(However, Alloy currently has no inference rules that take integrity constraints or negation into account.)

Positive introspection (4) This is straightforward:

$$A \vdash \ulcorner B \vdash \ulcorner B \vdash \dot{\dot{p}} \urcorner \urcorner \leftarrow \ulcorner B \vdash \dot{q} \urcorner \wedge p \; Names \; q$$

Negative introspection (5) If Alloy were to be extended with negation, then negative introspection is also easy:

$$A \vdash \ulcorner B \vdash not \ulcorner B \vdash \dot{\dot{p}} \urcorner \urcorner \leftarrow not \ulcorner B \vdash \dot{q} \urcorner \wedge p \; Names \; q$$

6.3 Program Composition Operators

Brogi, Mancarella, Pedreschi and Turini have proposed an algebra of operators for composing logic programs [9]. The operators are $P \cup Q$, $P \cap Q$, P^* and $P \triangleleft Q$, for union, intersection, encapsulation and import of programs, respectively. Their meta-interpretive definition can be coded elegantly in Alloy, provided that we choose one unary and three binary function symbols for constructing theory terms that stand for the theories resulting from these operations.

We let all the theories of a logic program with theory operators constitute a theory system internal to a theory M. The definition of M contains five theoremhood statements that define the theorems of theories named

by operator expressions. Here we represent the operators by the function symbols U, I, E and T, respectively.

$$M \vdash \ulcorner U(\dot{p}, \dot{q}) \vdash \dot{a} \leftarrow \dot{c} \urcorner \leftarrow \ulcorner \dot{p} \vdash \dot{a} \leftarrow \dot{c} \urcorner$$
$$M \vdash \ulcorner U(\dot{p}, \dot{q}) \vdash \dot{a} \leftarrow \dot{c} \urcorner \leftarrow \ulcorner \dot{q} \vdash \dot{a} \leftarrow \dot{c} \urcorner$$
$$M \vdash \ulcorner I(\dot{p}, \dot{q}) \vdash \dot{a} \leftarrow \dot{c} \urcorner \leftarrow \mathit{Partition}(c, c_1, c_2) \wedge \ulcorner \dot{p} \vdash \dot{a} \leftarrow \dot{c_1} \urcorner \wedge \ulcorner \dot{q} \vdash \dot{a} \leftarrow \dot{c_2} \urcorner$$
$$M \vdash \ulcorner E(\dot{p}) \vdash \dot{a} \leftarrow \mathit{True} \urcorner \leftarrow \ulcorner \dot{p} \vdash \dot{a}? \urcorner$$
$$M \vdash \ulcorner T(\dot{p}, \dot{q}) \vdash \dot{a} \leftarrow \dot{c_1} \urcorner \leftarrow \ulcorner \dot{p} \vdash \dot{a} \leftarrow \dot{c} \urcorner \wedge \mathit{Partition}(c, c_1, c_2) \wedge \ulcorner \dot{q} \vdash \dot{c_2}? \urcorner$$

This straightforward program, which uses a ground representation, is no less elegant than the program by Brogi & Contiero [11] that uses a nonground representation. (We assume that the ternary predicate *Partition* has been defined to compute the partition of a conjunction into a pair of (possibly empty or unitary) conjunctions.

For example, consider a program in the algebra with three "basic" theories *Rules*, *Public* and *Private* [10]. In the Alloy program, the clauses of these theories should appear as theoremhood statements $M \diamond \mathit{Rules} \vdash \cdots$, $M \diamond \mathit{Public} \vdash \cdots$, and $M \diamond \mathit{Private} \vdash \cdots$, respectively. We can then add a coincidence statement such as $\mathit{GiveCredits} \equiv M \diamond U(T(\mathit{Rules}, \mathit{Private}), \mathit{Public})$ in order to define a theory *GiveCredits* which can subsequently be queried. Any query to *GiveCredits* will then be computed in the composed theory $(\mathit{Rules} \triangleleft \mathit{Private}) \cup \mathit{Public}$.

6.4 Implicit Programming

Essentially all programs today are written manually by programmers. The programmers build on past experience and sometimes even directly on programs written in the past. (Indeed, this happens every time an existing program needs modification; we may see it as writing a program that is to perform almost the same computation as an existing program.) This might happen in many ways. Sometimes a program piece can be reused as is, when the abstraction it provides is exactly the one sought for. Typically pieces of the existing program need to be systematically rewritten in some way, for example, an extra argument might need to be added to a procedure or a base case replaced. If the existing program needs extensive rewriting, perhaps only its basic structure remains, such as the recursion pattern.

When really done systematically, this is a useful methodology. If the existing program does what is expected from it and each small change transforms it in a known way, then we may have confidence that the program resulting from a sequence of such changes computes what we expect. It is a serious problem today that modifications of the kind outlined above can rarely be carried out flawlessly. The resulting program then does not do what is expected and expensive corrective work is required. We may never know when the program becomes error-free.

Suppose we could partly automate this process, so the programmer could instead take a program or a program fragment and specify exactly which modifications must be done. The requested transformation would then be applied and the process continued until the desired program had been created. Given a collection of generally useful program fragments, the programmer might even build an entirely new program by incorporating and transforming these components. An alternative, often discussed in the realm of functional programming, is to provide very powerful abstractions so every problem can be coded in terms of these high-level abstractions. This approach is mathematically very appealing but has not yet turned out to be a practical approach to programming. The process outlined above is closer to an approach taken by actual programmers and also seems to be useful for reasoning exactly about the resulting programs. A more detailed comparison between these approaches seems necessary in the future.

As an example, let us show a simple program that adds an extra argument to a predicate. The transformation program is in a theory called T.

$$
\begin{aligned}
T \vdash \ulcorner \mathit{Extend}(\dot{m}, \dot{p}) \vdash \dot{b} \leftarrow \dot{d} \urcorner \leftarrow\ & \\
& \ulcorner \dot{m} \vdash \dot{a} \leftarrow \dot{c} \urcorner \wedge \\
& \mathit{NonoccurringVariable}(\ulcorner \dot{a} \leftarrow \dot{c} \urcorner, v) \\
& \mathit{TransAtom}(a, b, p, v) \wedge \\
& \mathit{TransQuery}(c, d, p, v) \wedge \\
T \vdash\ & \mathit{TransAtom}(\ulcorner \dot{p}(|\dot{x}) \urcorner, \ulcorner \dot{p}(\dot{v}|\dot{x}) \urcorner, p, v) \leftarrow \mathit{True} \\
T \vdash\ & \mathit{TransAtom}(\ulcorner \dot{q}(|\dot{x}) \urcorner, \ulcorner \dot{q}(|\dot{x}) \urcorner, p, v) \leftarrow p \neq q \\
T \vdash\ & \mathit{TransQuery}(\ulcorner \mathit{True} \urcorner, \ulcorner \mathit{True} \urcorner, \_, \_) \leftarrow \mathit{True} \\
T \vdash\ & \mathit{TransQuery}(\ulcorner \dot{a} \wedge \dot{c} \urcorner, \ulcorner \dot{b} \wedge \dot{d} \urcorner, p, v) \leftarrow \\
& \mathit{TransAtom}(a, b, p, v) \wedge \\
& \mathit{TransQuery}(c, d, p, v)
\end{aligned}
$$

(We assume that the predicate *NonoccurringVariable* has been defined to compute (in its second argument) some variable name that does not occur in the name given as the first argument.)

In order to use this program, we must make T's view of the inspected and the defined theories coincide with the actual theories that we wish to inspect and define:

$$
\begin{aligned}
& T \diamond \mathit{JohnsBrain} \equiv \mathit{JohnsOldBrain} \\
& T \diamond \mathit{Extend}(\mathit{JohnsBrain}, \mathit{Likes}) \equiv \mathit{JohnsNewBrain}
\end{aligned}
$$

Henceforth, the theory *JohnsNewBrain* will be exactly like the theory *JohnsOldBrain*, except that any clause which contains a predication $\mathit{Likes}(\cdots)$ has been replaced by a clause in which these predications have all been replaced by $\mathit{Likes}(\cdots, v)$, where v is some variable that did not occur in the original clause.

7 Self-reference

The reader may have noted that we have avoided using any circular theories. There is no automatic mechanism which gives a theory access to information about its own provability.

There are several advantages with systems that do not contain self-referring theories, i.e., theories that do not really reflect upon themselves but at most upon "views" of themselves. For example, there will be no paradoxes and implementation becomes simpler and more efficient.

The disadvantage with prohibiting or avoiding self-reference is, of course, a reduced expressivity. It is not possible to define agents that truly introspect. It is an open question at this time how serious a restriction it would be to prohibit self-reference completely, but it is clear that one can often make do with a sufficiently high tower of theories, each being a meta-theory for the theories below it. A very close approximation to a single theory which is a meta-theory for itself is obtained through an infinite tower of identical theories, each being a meta-theory for the theories below it. Such a tower can be expressed in Alloy.

If we wished to make an Alloy theory $T \diamond U$ truly self-referential, e.g., through a theory I in $T \diamond U$, we could add one of the two equivalent statements $T \diamond U \diamond I \equiv T \diamond U$ and $T \vdash \ulcorner U \diamond I \equiv U \urcorner$ to the program. It is easy to show that with either statement, in any model $\langle \mathcal{M}, \equiv \rangle$ of the program, we will have that $\mathfrak{M}_{T \diamond U \diamond I} \equiv \mathfrak{M}_{T \diamond U}$. That is, whatever $T \diamond U$ "observes" in the theory it calls I, is really also in $T \diamond U$ itself. This is a "two-way" self-reference: $T \diamond U$ may query itself by querying the theory it calls I, or it may compute clauses and add them to itself if it contains clauses such as $\ulcorner I \vdash \dot{p} \urcorner \leftarrow \cdots p \cdots$.

One could allow $T \diamond U$ to query itself but not add clauses to itself by instead adding here are three simple (and equivalent) ways: a theoremhood statement

$$T \vdash \ulcorner U \diamond I \vdash \dot{p} \urcorner \leftarrow \ulcorner U \vdash \dot{p} \urcorner,$$

to the program. It is easy to show that $\mathfrak{M}_{T \diamond U} \subseteq \mathfrak{M}_{T \diamond U \diamond I}$, i.e., that whatever is satisfied by $\mathfrak{M}_{T \diamond U}$ is also satisfied by $\mathfrak{M}_{T \diamond U \diamond I}$, so $T \diamond U \diamond I$ includes an "image" of $T \diamond U$.

However, note that there is no clause that could be added to $T \diamond U$ in order to achieve this effect. The rationale is simply that self-reference must be "sanctioned" from outside a theory.

8 Abduction

Abduction is a form of reasoning with a purpose to determine hypotheses that explain an observation, typically in the context of knowledge assimilation [23, 26]. Abductive reasoning seems particularly interesting in combination with meta-reasoning. Suppose the beliefs of John are represented by

a theory *Beliefs*(*John*), which internally defines a theory system in which there is a theory *Beliefs*(*Mary*), representing John's beliefs about Mary's beliefs. Suppose further that

$$\begin{array}{l} \mathit{Beliefs}(\mathit{John}) \vdash \\ \quad \mathit{SmilesAt}(a,b) \leftarrow \ulcorner \mathit{Beliefs}(\dot{u}) \vdash \mathit{Likes}(\dot{u},\dot{v})\urcorner \wedge u \ \mathit{Names}\ a \wedge v \ \mathit{Names}\ b, \end{array}$$

i.e., a statement that those who believe they like him smile at him. If John notices Mary smiling at him, we can assume that a belief *SmilesAt*(*Mary*, *John*) appears among John's beliefs, calling for an explanation. By performing abductive reasoning, the hypothesis $\ulcorner \mathit{Beliefs}(\mathit{Mary}) \vdash \mathit{Likes}(\mathit{Mary},\mathit{John})\urcorner$ appears as a good candidate for inclusion in *Beliefs*(*John*) because it would imply the observation. John therefore might assume that Mary believes she likes him.

This is of course merely a simple example but the area of agents performing meta-reasoning about each other's actions, beliefs, motives and ambitions is clearly one where abductive reasoning needs to be carried out as part of the meta-reasoning.

Abductive reasoning can be carried out in many ways. One way is to add inference rules for abductive reasoning, obtaining new abductive proof procedures [21]. However, it is also possible to realize abductive reasoning through meta-level deduction, as suggested by Bowen & Kowalski [8]. Such achievement of abductive reasoning through meta-reasoning is a topic that ought to be explored further using theory systems.

9 Implementation and language extensions

In our implementation efforts, we are extending Luther [5], an instance of Warren's abstract Prolog machine [32]. The idea is that the generalized SLD-resolution rule should be essentially as efficient as in Prolog, regardless of the number of "indirection" levels. This can be made possible by representing the clauses of all theories, also those that only exist as a "view" in some other theory, by ordinary abstract machine code. An interesting difficulty is when a program clause is not an explicit axiom in a theory but is obtained through some computation in a meta-theory of the current theory. This we intend to solve by never actually creating the program clause but rather use directly the parts of the program clause that are explicit in the meta-theory and then carry out a computation in the meta-theory. The following example should illustrate the technique. Consider the following program fragment.

$$\begin{array}{l} T_M \diamond T_O \equiv T_O \\ T_M \vdash \ulcorner T_O \vdash P(\dot{x}, F(\dot{y})) \leftarrow Q(\dot{y}) \wedge \dot{z}\urcorner \leftarrow R(\dot{x},\dot{y},\dot{z}) \end{array}$$

If we were to prove a P atom in T_O (or in $T_M \diamond T_O$), we could first carry out a computation in T_M of the complete name of some clause $P(\cdots, F(\cdots)) \leftarrow$

$Q(\cdots) \wedge \cdots$ where the dotted parts were filled in by the R atom in T_M. However, computing the whole clause could well be a waste of resources, as is easy exemplified: Suppose that the goal atom is actually $P(42, G(54))$. Unification of the goal atom with the head of the generated program clause will always fail immediately and the computation in T_M of the program clause would be worthless. What we do instead is to compile as part of the code reachable from T_O a clause

$$P(x_1, F(y_1)) \leftarrow x \; \mathit{Names} \; x_1 \wedge y \; \mathit{Names} \; y_1 \wedge R(x, y, z) \wedge Q(y_1) \wedge \alpha[z].$$

We see that all parts of the clause that were explicitly given in the meta-level clause are present in this clause. The two *Names* atoms constrain the variables x and y so that any value they obtain must be a name of something that can be unified with x_1 and x_2, respectively. The expression $\alpha[z]$ can best be described as a call to whatever becomes the value of z. In the worst case, this might require using an interpreter but it seems to us that in this situation, the value of z is usually taken from some context where there is machine code available for the named query. In this case, that code can be used (with some care). If we consider again the goal atom $P(42, G(54))$ we see that this clause will fail before computing any part of the body.

As mentioned before, the style of computation described above realizes a different inference system from the one described in Sect. 4.3. In this system, computations in various theories can be interleaved, as shown by the example. The idea is to be as goal-directed as possible.

It is clear that negation of some kind must be added to the language, either explicit negation, negation as failure or both. If we incorporate negation as failure in Alloy, we will investigate the merits of a monotonic version of negation as failure, where the theory in which a finitely failed proof is obtained is given explicitly.

It would also be very interesting to incorporate some form of abductive procedure in Alloy, because of the natural links between meta-reasoning and abduction pointed out in Sect. 8. Denials are already formally present in the language and would then function as integrity constraints when given as part of a program [26].

10 Notes and related work

There have been a few changes in the definition of Alloy since our previous publication [4].

1. Theory terms now include expressions on the form $\cdots \diamond \cdots$.
2. In addition to program clauses, Alloy now has goals and denials.
3. What used to be called a tagged program clause is now called a theoremhood statement and may contain any sentence.

4. Representation statements have been generalized to coincidence statements (a representation statement $t \triangleright u$ can be written as $t \diamond u \equiv u$). This allowed us to generalize the reflection rules and simplify the inference system considerably.

5. There is an SLD-resolution style inference rule instead of an inference rule for program clauses.

It should be obvious for the knowledgeable reader that the development of Alloy is very much inspired by work of Kowalski [25, 26], and by Reflective Prolog of Costantini & Lanzarone [16].

There have recently appeared some proposals for systems for meta-reasoning with a similar philosophy as ours. Attardi & Simi [1] use what they call "relativized truth" but obtain a system quite similar to ours. One significant difference is that they choose to duplicate their inference system (a natural deduction system): the rules are present once for the object level and again for the meta-level. Moreover, among their basic axioms for the meta-level, there is one which ensures positive introspection. We have preferred to have no such epistemic bias, except for saturation. Giunchiglia *et al.* [18] have defined a multilevel deduction system with distinct levels, called MK. There is only one theory per meta-level but the communication between meta-levels is similar to that in Alloy. This seems to be the basis for the reasoning part of `GETFOL`, a system that is also capable of code introspection and revision [17].

Our proposal for a meta-programming based software engineering methodology is related to the proposal by Kowalski about using meta-language for assembling programs [24] and the work by Brogi *et al.* about using theory operators for building programs, which is discussed in more detail in Sect. 6.3.

Bowen & Weinberg [6] and Bacha [2] have investigated compilation of partially known clauses in a context similar to ours.

Sato [29] proposes an approach to meta-programming through a complete truth predicate *tr* in three valued logic. Sato's definition of *tr* is self referential, and gives in the general case an inconsistent definition of *tr* in two valued logic by being paradoxical. As a slight modification of the definition of *tr* he introduces a three valued complete *demo* predicate.

The language is fully amalgamated, like the theory part of Alloy to which it corresponds. (Note, however, that the system part and the theory part of Alloy are clearly separated both syntactically and semantically.)

The main similarity with our approach to meta-programming is the ability to reason with several levels, which is made possible by *tr* being self-referential; thus making it possible to express $tr(\ulcorner \ldots tr(\ldots,\ldots)\ldots \urcorner,\ldots)$ (the nesting can be of arbitrary depth). Furthermore, like naming in Alloy,

the structural coding makes it possible to decompound terms and formulas to their least parts and look, for example, at codes of functors.

Jiang [20] proposes an ambivalent approach to meta-reasoning, by introducing a language called AL where syntactically no distinction is made between terms, formulas or functors. Jiang takes a radically different approach from ours by defining what he calls a "Herbrand-based" semantics, which does not build upon the standard notion of logical consequence in first order model theory. It is hard to form a definitive opinion of the proposed semantics because as it is presented, it is not well-defined and thus cannot be understood without having to guess the intentions. Neither does he present an inference system, nor hint at any possible implementation of the proposed ideas. (It should be noted, however, that AL to some extent captures meta-programming as it is often done in Prolog, which has an operational semantics.)

Syntactically, the main similarity with our approach is the possibility to express reasoning across several meta-levels. The main distinction is that there is no naming or coding involved, formulas can occur directly as subexpressions in other formulas. The program clause 1 could for example be expressed as

$$Bel(\,Tim, \forall x(\,Cannibal(x) \rightarrow Bel(x, Tasty(x))).$$

The idea is that whether an expression is to be interpreted as a function or a relation is determined by the context where it appears.

Christiansen has proposed an amalgamated language in which there are two levels of reasoning [13]. The operational semantics of the language is based on instance predicates, relating names of formulas such that one is an instance of the other. As was shown by Kowalski [25, 26] and further developed by Hill & Gallagher [19], such instance predicates can be used with meta-variables replacing names of subexpressions in a way which turns out to be operationally similar to the way in which variables are represented using nonground representations.

11 Conclusion

As can be seen from this article, Alloy is a language still under development. We can already conclude, however, that it allows a direct way of expressing multilevel knowledge, in particular recursive beliefs.

The main difference between Alloy and the mainstream of meta-logic programming lies in the support for arbitrary many meta-levels and in that self-reference is the exception rather than the rule.

One may certainly doubt that a language claimed to be so powerful is efficiently implementable and this can only be proved by an actual implementation, which is under way. One reason for hope is the belief that much

of the computation will still be deduction within a single theory (which may be someone's view of someone's view of ... a theory) and this should be possible to support with essentially the efficiency of an ordinary Prolog system. The difficulties seem to lie in the meta-programming specific parts and in the fact that there are so many ways to use a piece of information in a meta-programming setting. For example, a program clause may be actually used for deduction, a name for it may be used as data, so may a name for a name for it, etc. Program clauses computed from names with "holes" is likely to be another (manageable) obstacle to efficient computation.

Acknowledgements

This research has been influenced by valuable discussions with our colleagues, particularly Stefania Costantini, Gaetano Lanzarone, and Andreas Hamfelt, and our partners in the Compulog 2 project, particularly Antonio Brogi, Pat Hill, Bob Kowalski and John Lloyd.

The research reported herein was supported financially by the Swedish National Board for Technical and Industrial Development (NUTEK) under contract No. 92–10452 (ESPRIT BRP 6810: *Computational Logic 2*).

J. B. thanks his family for their continuing support.

References

[1] Attardi, G. and Simi, M., Building Proofs in Context, in: F. Turini (ed.), *Proc. META 94*, LNCS 883, Springer-Verlag, Berlin, 1994.

[2] Bacha, H., Meta-Level Programming: a Compiled Approach, in: J.-L. Lassez (ed.), *Proc. 4th Intl. Conf. on Logic Programming*, MIT Press, Cambridge, Mass., 1987.

[3] Barklund, J., Metaprogramming in Logic, UPMAIL Technical Report 80, Uppsala Univ., Computing Science Dept., 1994, to be published in *encyclopedia of computer science and technology*, marcel dekker, new york.

[4] Barklund, J., Boberg, K. and Dell'Acqua, P., A Basis for a Multilevel Metalogic Programming Language, in: F. Turini (ed.), *Proc. META 94*, LNCS 883, Springer-Verlag, Berlin, 1994.

[5] Bevemyr, J., The Luther WAM Emulator, UPMAIL Tech. Rep. 72, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1992.

[6] Bowen, K. A. and Weinberg, T., A Meta-Level Extension of Prolog, in: J. Cohen and J. Conery (eds.), *Proc. 1985 Symp. on Logic Programming*, IEEE Comp. Soc. Press, Washington, D.C., 1985.

[7] Bowen, K. A., Meta-Level Programming and Knowledge Representation, *New Generation Computing*, 3:359–383 (1985).

[8] Bowen, K. A. and Kowalski, R. A., Amalgamating Language and Metalanguage in Logic Programming, in: K. L. Clark and S.-Å. Tärnlund (eds.), *Logic Programming*, Academic Press, London, 1982.

[9] Brogi, A., Mancarella, P., Pedreschi, D. and Turini, F., Composition Operators for Logic Theories, in: J. W. Lloyd (ed.), *Computational Logic*, Springer-Verlag, Berlin, 1990.

[10] Brogi, A., Program Construction in Computational Logic, Ph.D. Thesis, Dipartimento di Informatica, Università di Pisa, 1993.

[11] Brogi, A. and Contiero, S., Gödel as a Meta-Language for Composing Logic Programs, in: F. Turini (ed.), *Proc. META 94*, LNCS 883, Springer-Verlag, Berlin, 1994.

[12] Brogi, A. and Turini, F., Metalogic for Knowledge Representation, in: J. A. Allen, R. Fikes and E. Sandewall (eds.), *Principles of Knowledge Representation and Reasoning: Proc. 2nd Intl. Conf.*, Morgan Kaufmann, Los Altos, Calif., 1991.

[13] Christiansen, H., Efficient and Complete Demo Predicates for Definite Clause Languages, Technical Report 51, Dept. of Computer Science, Roskilde University, 1994.

[14] Clark, K. L., Negation as Failure, in: H. Gallaire and J. Minker (eds.), *Logic and Data Bases*, Plenum Press, New York, 1978.

[15] Costantini, S., Dell'Acqua, P. and Lanzarone, G. A., Extending Horn Clause Theories by Reflection Principles, in: C. MacNish, D. Pearce and L. M. Pereira (eds.), *Logics in Artificial Intelligence*, LNAI 838, Springer-Verlag, Berlin, 1994.

[16] Costantini, S. and Lanzarone, G. A., A Metalogic Programming Language, in: G. Levi and M. Martelli (eds.), *Proc. 6th Intl. Conf. on Logic Programming*, MIT Press, Cambridge, Mass., 1989.

[17] Giunchiglia, F. and Cimatti, A., Introspective Metatheoretic Reasoning, in: F. Turini (ed.), *Proc. META 94*, LNCS 883, Springer-Verlag, Berlin, 1994.

[18] Giunchiglia, F., Serafini, L. and Simpson, A., Hierarchical Meta-Logics: Intuitions, Proof Theory and Semantics, in: A. Pettorossi (ed.), *Meta-Programming in Logic*, LNCS 649, Springer-Verlag, Berlin, 1992.

[19] Hill, P. M. and Gallagher, J., Meta-Programming in Logic Programming, Technical Report 94.22, School of Computer Studies, Univ. of

Leeds, 1994, to be published in *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 5, Oxford Science Publ., Oxford Univ. Press.

[20] Jiang, Y., Ambvivalent Logic as the Semantic Basis of Metalogic Programming, in: P. Van Hentenryck (ed.), *Logic Programming, Proc. 11th Intl. Conf*, MIT Press, Cambridge, Mass., 1994.

[21] Kakas, A. C. and Mancarella, P., Abductive Logic Programming, in: *Proc. NACLP90 Workshop on Non-Monotonic Reasoning and Logic Programming*, MCC, Austin, Texas, 1990.

[22] Konolige, K., *A Deduction Model of Belief*, Pitman, London, 1986.

[23] Kowalski, R. A., *Logic for Problem Solving*, North Holland, New York, 1979.

[24] Kowalski, R. A., The Use of Metalanguage to Assemble Object Level Programs and Abstract Programs, Report, Imperial College, London, 1982.

[25] Kowalski, R. A., Meta Matters, Invited presentation at Second Workshop on Meta-Programming in Logic, 1990.

[26] Kowalski, R. A., Problems and Promises of Computational Logic, in: J. W. Lloyd (ed.), *Computational Logic*, Springer-Verlag, Berlin, 1990.

[27] McCarthy, J., Programs with Common Sense, in: M. Minsky (ed.), *Semantic Information Processing*, MIT Press, Cambridge, Mass., 1968.

[28] McCarthy, J., First Order Theories of Individual Concepts and Propositions, in: B. Meltzer and D. Michie (eds.), *Machine Intelligence 9*, Edinburgh University Press, Edinburgh, 1979.

[29] Sato, T., Meta-Programming through a Truth Predicate, in: K. Apt (ed.), *Proc. Joint Intl. Conf. Symp. on Logic Programming 1992*, MIT Press, Cambridge, Mass., 1992.

[30] Smorynski, C., The Incompleteness Theorems, in: J. Barwise (ed.), *Handbook of Mathematical Logic*, North-Holland, Amsterdam, 1977.

[31] Sterling, L. S., Logical Levels of Problem Solving, *J. Logic Programming*, 1:138–45 (1984).

[32] Warren, D. H. D., An Abstract Prolog Instruction Set, SRI Tech. Note 309, SRI Intl., Menlo Park, Calif., 1983.