TPLP: Page 1–41. © The Author(s), 2026. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.
doi:10.1017/S1471068426100465

Multi-paradigm Logic Programming in the ℰrgoAI System[*]

MICHAEL KIFER[†]
Computer Science, Stony Brook University, USA
(*e-mail:* kifer@cs.stonybrook.edu)

THERESA SWIFT
Coherent Knowledge Systems, Arlington, USA
(*e-mail:* theresasturn@gmail.com)

submitted 21 March 2025 UTC; revised 13 November 2025 UTC; accepted 8 April 2026 UTC

Abstract

ℰrgoAI is a high-level, multi-paradigm logic programming language and system developed by Coherent Knowledge Systems as an enhancement of and a successor to the popular *ℱlora-2* system. ℰrgoAI is oriented toward scalable knowledge representation and reasoning and can exploit = structured knowledge as well as knowledge derived from external sources such as vector databases. From their inception, ℰrgoAI (and *ℱlora-2* before it) were designed to exploit the well-founded semantics for reasoning in a multi-paradigm environment, including *object-based* logic (F-logic) with non-monotonic inheritance; *higher-order* syntax in the style of HiLog; *defeasibility* of rules; semantically clean *transactional updates*; *subgoal delay* for handling unsafe queries and for better performance; and optional support for bounded rationality at a module level. Although ℰrgoAI programs are compiled into XSB and adopt many Prolog features, ℰrgoAI is altogether a different language and system. Under consideration in Theory and Practice of Logic Programming (TPLP).

KEYWORDS: non-monotonic reasoning, frame logic, transaction logic, tabling

1 Introduction

ℰrgoAI is an open-source knowledge representation and reasoning (KRR) system oriented toward scalable reasoning about hybrid knowledge in a dynamically changing environment. ℰrgoAI is the successor to and replacement for the well-known *ℱlora-2* KRR

[*] We are grateful to Benjamin Grosof and Paul Fodor for many technical and visionary discussions. Many of the additions to ℰrgoAI were inspired by this collaboration. We thank Daniel Elenius for suggesting an idea that gave rise to the `RefuteCLP` argumentation theory discussed in Section 6. We are also grateful to Martin Gebser and Torsten Schaub for their comments on the suitability of various problems for ASP and tabled Prolog.
[†] Supported in part by the NSF grant 1814457.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

system Yang *et al.* (2003); Kifer (2022), which is no longer maintained. $\mathcal{E}$rgoAI includes many features either not in, or much improved from those in $\mathcal{F}$*lora-2*.[1]

The language of $\mathcal{E}$rgoAI is based on HiLog (*higher-order* syntax) Chen *et al.* (1993) and Frame logic (abbr. F-logic) supporting objects, types, inheritance Kifer *et al.* (1995) set within a strong dynamic module system. In addition, $\mathcal{E}$rgoAI cleanly integrates general formulas with the *exists* and *forall* quantifiers; user-defined functions; *defeasible* reasoning in the LPDA style Wan *et al.* (2009); *transactional updates* Bonner and Kifer (1994) with integrity constraints and alerts; bounded rationality Grosof and Swift (2013); Riguzzi and Swift (2014); and dynamic subgoal reordering. $\mathcal{E}$rgoAI has been designed so that these extensions fit together hand in glove, allowing for a high degree of coherence both in language and semantics.

$\mathcal{E}$rgoAI treats applications written in other languages, such as Prolog, Python, and C, simply as special modules, which enables tight integration. For instance, $\mathcal{E}$rgoAI can use knowledge available in Python, such as vector embeddings, in combination with traditionally structured knowledge. Due to space limitations, this paper covers only some of the aforesaid paradigms.

Unlike KRR systems, such as ASP solvers and description logic provers, $\mathcal{E}$rgoAI is based on the well-founded semantics (WFS) van Gelder *et al.* (1991) which has low-order polynomial data complexity. By choosing a weaker logic, $\mathcal{E}$rgoAI gains several advantages. For instance, $\mathcal{E}$rgoAI evaluations do not need to be grounded: as will be shown, even non-ground negation is soundly evaluated. The lack of need for grounding allows $\mathcal{E}$rgoAI to use unrestricted logical terms as data structures – a use that distinguishes $\mathcal{E}$rgoAI from Datalog systems. However $\mathcal{E}$rgoAI shares with Datalog (and Prolog) the ability to evaluate queries in a relevant, top-down manner.

Although $\mathcal{E}$rgoAI programs are compiled into XSB Prolog Swift and Warren (2012), $\mathcal{E}$rgoAI relies on its own reader, optimizing compiler, and runtime libraries.[2] At the same time, $\mathcal{E}$rgoAI relies heavily on XSB's tabled evaluation and directly benefits from XSB's close connection with Python through the Janus interface Swift and Andersen (2023).

$\mathcal{E}$rgoAI demonstrates that a variety of new logical features can be incorporated into a fast, robust, and scalable system. For instance, in $\mathcal{E}$rgoAI all predicates and frames support HiLog by default, so that the functor of a term or predicate can be any (not necessarily ground) term, as opposed to an atomic functor, as in Prolog. Similarly, although Prologs such as XSB, SWI, Ciao, and YAP support tabled resolution, they use SLDNF resolution as a default. In $\mathcal{E}$rgoAI the *default* resolution method is SLG resolution for WFS, extended with (*R*)eactive incremental tabling, and with optional bounded rationality through subgoal and answer (*A*)bstraction, a resolution method we term *SLGR(A)*. If desired though, a predicate P can be declared as non-tabled if P contains side effects, or for other purposes.

The journey from Flora-2 to $\mathcal{E}$rgoAI began in 2013 when a small group formed Coherent Knowledge Systems to develop Flora-2 into a more robust, commercially supported system. $\mathcal{E}$rgoAI extends Flora-2 in having an IDE, explanation capabilities, full reactive

[1] This paper is a significant extension of Swift and Kifer (2024).
[2] A rough measure of the sophistication of $\mathcal{E}$rgoAI is that it is based on over 200,000 lines of Prolog, $\mathcal{E}$rgoAI, and C code, while its manuals have a length of over 600 pages.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

tabling, instantiation delay, termination support, and tripwires, as well as many smaller features. However, perhaps the most effort was spent to make ℰrgoAI more robust and scalable both in ℰrgoAI's runtime libraries and at the XSB level. As a result of ℰrgoAI's features and stability, the system is being used for research and commercial applications in software assurance Bloomfield (2024), financial compliance, tax accounting, and legal reasoning.[3] In addition, it is currently being used in the DARPA CODORD project as a target language for the generation of executable logic from English text CODORD (2024).[4]

This paper describes a few of ℰrgoAI's logical extensions at a technical but accessible level. The first group of *non-monotonic* extensions includes defeasible inheritance, the use of default and explicit negation, and argumentation theories that are adjoined to programs to provide different flavors of defeasibility. Non-monotonic features make use of the truth value u, which represents truth values that are either *undefined* in WFS or that are *unknown* due to the use of bounded rationality. The second group of *dynamic* capabilities includes *reactive tabling* that automatically updates inferences when data changes, along with integrity constraints that signal and take actions whenever updated inferences fulfill specified conditions. These dynamic capabilities work within the framework of *transaction logic* to ensure that each update is semantically meaningful. A third *query evaluation* group (1) minimizes floundering of unsafe goals (including negation) through *instantiation delay* and (2) supports sound termination of *any* program through a type of bounded rationality called *restraint*.

The paper is structured as follows. Section 2 provides background on WFS and *SLGR(A)*, ℰrgoAI's tabled resolution method. Sections 3 and 4 describe the base language of ℰrgoAI: frame-logic with inheritance and HiLog.[5] Section 5 overviews ℰrgoAI's module system, which allows code files to be associated with modules in a completely dynamic manner. Section 6 describes ℰrgoAI's default argumentation theory for defeasible reasoning and Section 7 its approach to soundly evaluate unsafe goals. Section 8 discusses ℰrgoAI's approach to KRR in a dynamically changing environment, while Section 9 discusses ℰrgoAI's approaches to termination including sound termination through bounded rationality. Section 10 provides a summary of ℰrgoAI's architecture. Finally, Section 11 provides results on scalability along with comparisons of ℰrgoAI's speed and memory consumption to that of Prolog.

When reading the following sections that describe ℰrgoAI features (Sections 2–9) users may wish to refer to Section 10, which provides a high-level summary and diagram of ℰrgoAI's architecture.

2 Preliminaries: The WFS and tabling

In the late 1980s the semantics of negation in logic programming was an active topic of research from which two main semantics emerged: the stable model semantics Gelfond and Lifshitz (1988) and WFS van Gelder *et al.* (1991). These semantics are related, as the

[3] The website for Coherent Knowledge Systems http://coherentknowledge.com lists a number of applications in the above areas. Also, see the TED talk on legal reasoning which contains a discussion of legal modeling with ℰrgoAI Morris (2018).
[4] ℰrgoAI is open-source and available at `https://github.com/ErgoAI`.
[5] This language is sometimes termed *Rulelog* Andersen *et al.* (2013).

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

```
P_reach :
  reachable(?X,?Y):- reachable(?X,?Z),edge(?Z,?Y)
  reachable(?X,?Y):- edge(?X,?Y)
  edge(1,2).     edge(2,3).     edge(3,1).
```

Fig. 1. $\mathcal{E}$rgoAI program to illustrate positive recursion handling in tabling.

well-founded model of a program P is contained within the intersection of every stable model of P Przymusinski (1990) and so is a total stable model in the case of programs with stratified negation, and a partial stable model otherwise. The stable model semantics has given rise to the important field of Answer Set Programming (cf. Marek and Truszczyński (1999)). WFS however, remains important despite the fact that it is a weaker logic than that of stable models. WFS provides a semantics for *any* program including those with function symbols or those that have no stable model. WFS also has a low-polynomial data complexity van Gelder (1989). This lower complexity means that (1) programs do not need to be grounded to be evaluated;[6] and (2) when tabled evaluation (tabling) is used, a query Q can be evaluated in a top-down manner using only relevant clauses.

Before describing WFS, consider first the behavior of SLDNF (cf. Lloyd (1984)), the resolution method used by Prolog. If negation is not used, SLDNF resolution is simply Horn Clause resolution with a fixed literal selection strategy, which in Prolog is always left to right.[7] SLDNF also supports a type of *negation as failure* via the Prolog operator + (called `\naf`, in $\mathcal{E}$rgoAI syntax). However, SLDNF will not terminate for the query "`?- s.`" to any of the programs in Figures 1 or 2, all of which contain positive and/or negative loops.

In contrast to SLDNF, $\mathcal{E}$rgoAI fundamentally relies on *SLGR(A)* resolution, which evaluates WFS via tabled *SLG resolution* Chen *et al.* (1995); Chen and Warren (1996) extended to provide *R*eactivity to updates, and – optionally – *A*bstractions for bounded rationality. In this section we focus on SLG, which $\mathcal{E}$rgoAI uses to support WFS as well as to adjoin argumentation theories to a program (Section 6). Reactivity of tabling is discussed in Section 8, while tabling with bounded rationality is discussed in Section 9.

In fact, $\mathcal{E}$rgoAI automatically uses tabled evaluation for queries to any predicate not marked as transactional.[8] The formal operational semantics of SLG has been presented in numerous papers (cf. Swift and Warren (2012) for an overview); in this paper we present tabling informally, in order to maintain focus on $\mathcal{E}$rgoAI. SLG resolution includes three mechanisms necessary to evaluate queries according to WFS: *loop detection*, *incremental completion* for sets of mutually dependent subgoals, and an *adaptive literal selection strategy*.[9]

We illustrate loop detection through evaluation of the query `reachable(1,?Node)` to the definite Datalog program $\mathbf{P}_{reach}$ in Figure 1. $\mathbf{P}_{reach}$ is in $\mathcal{E}$rgoAI's *predicate syntax*, a subset of its HiLog syntax, as explained in Section 4.

[6] Section 7.1 discusses how $\mathcal{E}$rgoAI addresses non-ground negative literals during an evaluation.
[7] A *literal selection strategy* refers to the order of resolution for literals in the body of a rule.
[8] Since SLDNF is the default evaluation strategy in Prologs that support tabling, a tabling declaration is required to ensure termination and correctness for the query to $\mathbf{P}_{reach}$.
[9] To fully support WFS, SLG also performs other operations such as simplification and answer completion that are not discussed here.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

$\mathbf{P}_{\text{unfounded}_1}$:

```
s:- \naf p,\naf q,\naf r.      p:- q,\naf r.
q:- r,\naf p.                  r:- p,\naf q.
```

$\mathbf{P}_{\text{undefined}}$:

```
s:- \naf p,\naf q,\naf r.      p:- \naf r.
q:- \naf p.                    r:- \naf q.
```

$\mathbf{P}_{\text{unfounded}_2}$:

```
s:- \naf p,\naf q,\naf r.      p:- \naf r,q.
q:- \naf p,r.                  r:- \naf q,p.
```

Fig. 2. ErgoAI programs to illustrate WFS and SLG.

This syntax is similar to that of Prolog except that variables in ErgoAI are denoted by identifiers that begin with `?` rather than with an upper case letter.

Whenever SLG evaluates a subgoal S, it checks whether a variant of S is already present as a table entry.[10] If not, as when S is the top-level goal, a table entry for S is created, program clauses are used for resolution, and any derived answers will be added to the table entry for S. Otherwise, if a table entry already exists, as when the goal `reachable(1,?Z)` is encountered as a body literal, the evaluation sets up a *consumer subgoal* for S, and uses the tabled answers to resolve against S. By guaranteeing that all answers for a subgoal S are returned to all consumer subgoals for S, tabling's loop detection is sufficient to ensure correctness for queries to programs like $\mathbf{P}_{reach}$ that have no negation, but do have the *bounded term size* property in which all subgoals and answers have a bounded size.

For programs that contain negation such as $\mathbf{P}_{unfounded_1}$ in Figure 2, goal-directed evaluation of a query such as `s` also requires *incremental completion*. To support incremental completion, XSB's tabling engine implicitly maintains a dependency graph, G_{dep}, among *active* subgoals. G_{dep} is a signed, directed graph in which there is an edge from subgoal S_1 to subgoal S_2: if S_1 (directly) *depends on* S_2: that is if there exists a substitution θ for a rule R in a program P where S_1 is a variant of the head of $R\theta$ and S_2 is a variant of the underlying subgoal of a literal L in the body of $R\theta$. The edge is signed as positive or negative depending on whether L is a positive or negative literal. A *strongly connected component (SCC)* in G_{dep} indicates a set of mutually dependent subgoals. In the evaluation of $\mathbf{P}_{unfounded_1}$, the set of mutually dependent subgoals, `p,q,r` forms an SCC. A tabling engine with incremental completion detects when all tabling operations have been performed for all subgoals in an SCC as well as for all dependent subgoals. At completion, the status of the subgoals in the SCC (`p,q,r`) is changed from *active* to *complete* and the SCC is removed from G_{dep}. Since no subgoal in this SCC has any answers, and since all dependencies within the SCC are positive, the subgoals form an *unfounded set*.[11] Their truth values are set to *false*, after which the subgoals `\naf p`, `\naf q` and `\naf r` can all be resolved leading to the success of `s`. Since $\mathbf{P}_{unfounded_1}$ has a

[10] Two terms are variant if they are equal up to variable renaming. ErgoAI uses a type of tabling called *call variance* but also has an experimental version that uses *call subsumption*, which can be faster for some goals.

[11] In WFS, infinite dependency chains also form unfounded sets. ErgoAI's approach to such unfounded sets will be discussed in Section 9.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

2-valued well-founded model and since all unfounded sets can be discovered using a fixed ordering of a literal selection strategy, $\mathbf{P}_{unfounded_1}$ is *fixed order dynamically stratified* Sagonas *et al.* (2000).

The next two programs in Figure 2, $\mathbf{P}_{undefined}$ and $\mathbf{P}_{unfounded_2}$, demonstrate the importance of SLG's flexible literal selection strategy. $\mathbf{P}_{undefined}$ is the same as $P_{unfounded_1}$ except that all positive literals have been removed from rule bodies. In this case, evaluation of the goal `s` again discovers that `p`, `q`, and `r` form an SCC with no answers derived for any of the goals. However, because there are negative dependencies among these subgoals, they do not form an unfounded set as before. In such a case, SLG *delays* evaluation of literals involving the negative cycle (`\naf p`, `\naf q`, and `\naf r`). Delayed literals are conceptually represented using a bar (`|`) in a partially evaluated rule body. Literals to the right of the bar have been delayed, while literals to the left of the bar have not yet been evaluated. The state of a tabled evaluation can be partially represented by the trees for each active subgoal. Since `p`,`q`, and `r` are all active, and since each active subgoal has a single rule, the state of the evaluation after these delays can be schematically represented as:

```
p:- \naf q|.    q:- \naf r|.    r:- \naf p|.
```

At this state, there are no other applicable operations for any of the three active subgoals, so SLG adds the answers below to the tables of the subgoals. In the operational semantics of SLG, answers like these that have a non-empty set of delayed literals correspond to atoms that are undefined in WFS with truth value u.

The third program, $\mathbf{P}_{unfounded_2}$ changes the order of positive and negative body literals from $\mathbf{P}_{unfounded_1}$. In this case, `p`, `q`, and `r` will again form an unfounded set, but each occurs to the right of a negative literal. As in $\mathbf{P}_{undefined}$, SLG handles this by delaying negative literals for the goals in the SCC. The state after these delays is represented by the partially evaluated clauses:

```
p:- \naf q|r.    q:- \naf r|p.    r:- \naf p|q.
```

Once the literals have been delayed, the evaluation continues, discovering the unfounded set. At this stage no operations are available for these subgoals, and they are completed. Since they are completed with no answers, SLG marks them as *false*, leading to the success of the goal `s` as in $\mathbf{P}_{unfounded_1}$.

Our purpose in this section is simply to review some aspects of SLG that are used later in the paper. Several complete descriptions of SLG can be found in the literature (cf. the references in the discussion of tabling in Swift and Warren (2012)). While the programs in Figure 2 might seem contrived, they represent the types of evaluations $\mathcal{E}$rgoAI must perform for its argumentation theories (Section 6). Finally, we note that $\mathcal{E}$rgoAI also uses extensions of SLG to evaluate transaction logic statements (Section 8) and to ensure sound termination of queries to programs with function symbols via bounded rationality (Section 9).

3 Frame logic

Frame Logic (abbreviated F-logic) Kifer *et al.* (1995) offers a number of benefits for representing knowledge using a style that is both object-oriented and ontology-oriented.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

```
person('Bunky Muntner')[age->63,
    residesIn->zip(20016)[city->Washington,state->DC],
    occupation->journalist,
    spouse->'Ginger Muntner',
    language->Spanish].

claim(13355)[dateTime->"2021-12-23T12:33:55"^^\dt,
    sourceDocument-> 'Jalapeno Springs Daily',
    medium -> NewsSite,
    sourceText -> 'My vitamin supplement cures everything.'
    statement -> statement1[disease->COVID,cure->takeSupplements ],
    embedding -> pyObj(p0x7faca3428510)].

  Person :: Agent.                Organization :: Agent.
person(?) : Person.               claim(?) : Claim.
  claim(13355) : NewsReport.
```

Fig. 3. ℰrgoAI instance frames with a fragment of a classification hierarchy.

Although ℰrgoAI can make use of external knowledge represented in nearly any format, a pure ℰrgoAI program consists of rules and facts formed from a mixture of HiLog predicates and F-logic *frame formulas* or *frames*. In this section we highlight frames along with *classification predicates* to denote a hierarchy of classes and instances and then describe how the two together provide structural and behavioral inheritance for a program. We should note here that the syntax used in ℰrgoAI differs from the original F-logic Kifer *et al.* (1995) in many respects. That original syntax was first improved by the *Flora-2* system Yang *et al.* (2003); Kifer (2022) and then further extended in ℰrgoAI.

Throughout this paper, ℰrgoAI features are illustrated using a set of examples motivated by the US DARPA AIDA project.[12] In 2021–2022, the AIDA project constructed sample problems centered around tracking misinformation about the COVID-19 virus NIST (2022). To address these problems, various natural language inputs were analyzed, and the results represented as a set of knowledge graphs over which various types of reasoning and hypothesis generation were performed.[13]

3.1 Frames and inheritance

Figure 3 shows two ℰrgoAI frames, each with an *object-id*, such as `person('Bunky Muntner')` or `claim(13355)`, followed by a list of attribute–value pairs of the form ⟨*attr*⟩ `->` ⟨*val*⟩. For instance, the types of the `claim` attribute values include atoms, an RDF XSD date type instance, another frame expression representing the semantic frame of a statement, and even the reference to a Python object. Figure 3 also shows a fragment of the classification hierarchy for the program, where the infix operator `::/2` represents the subclass relation and `:/2` indicates that an instance is a member of a class. Since their object-ids identify instances, these frames are called *instance frames*. Both `::/2` and `:/2` can be specified either by facts or rules. For instance, the non-ground fact on

[12] www.darpa.mil/program/active-interpretation-of-disparate-alternatives

[13] Names and organizations in the examples are motivated by the comic strip "Washingtoon" popular in the 1980s and 1990s.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

```
Claim[|dateTime{1..1}=>\dateTime,
       sourceDocument=>Document,
       documentType=>PublicWebDocument
       sourceText=>AtomicConstant,
       statement=> CompoundTerm,
       claimer=>Agent,
       embedding => PythonObject,
       medium->X|].
Claim[author->'Ingrid B. Baird'].

NewsReport[|documentType->WebNewsArticle|].
?Agt[claimed->?Claim]:- ?Agt:Agent,?Claim[claimer->?Agt].
WebNewsArticle::PublicWebDocument.
```

```
Person[|age=>\integer,
   spouse=>Person,
   residesIn=>Location,
   politAffil=>PolitParty,
   occupation=>Occupation,
   claimed=>Claim,
   language->English,
   politAffil->Independent|].
```

Fig. 4. $\mathcal{E}$rgoAI class signatures.

line 14 indicates that any frame with an object-id that unifies with `claim(?)` belongs to the class `Claim`.[14]

Figure 4 shows $\mathcal{E}$rgoAI class frames that specify *signatures* for subclasses and instances, default values for instance attributes, as well as attribute values of the class itself. When an attribute value is a type, the attribute–value relation is designated via `=>/2` and the inheritance is termed *structural*; otherwise `->/2` is used to designate default object values, and the inheritance is termed *behavioral*. Both type and object values are inheritable if they occur within [| . . . .|]. but otherwise are non-inheritable. For instance, in the `Claim` frame in Figure 4, all attribute values are inherited except for `author`, which pertains to the `Claim` class itself, and has the object `Ingrid B. Baird` as its value. Also in the frame for `Claim` the attribute `dateTime` is specified to be required and functional through a *cardinality constraint*, which indicates the minimum and maximum number of values an instance may have for a given attribute. The `Claim` frame also denotes the default object value `X` for the attribute `medium`. Finally, an $\mathcal{E}$rgoAI rule on line 12 propagates to person instances values of `claimed` from `claimer` attributes of claims.

The instances in Figure 3 inherit these attributes in various ways:

- *Direct behavioral inheritance.* Because the instance `'Bunky Muntner'` is a `Person`, it directly inherits `politAffil->Independent`.
- *Behavioral overriding.* The claim object `claim(13355)` states explicitly that `medium->NewsSite` (in Figure 3), overriding the default `medium->X` inherited from the class `Claim`.
- *Monotonic structural inheritance.* `claim(13355)` is a member of both the class `Claim`, from which it inherits the default `documentType` as `PublicWebDocument`, and of class `NewsReport` where `documentType` has the type `WebNewsArticle`. Unlike behavioral inheritance of attribute values, structural type inheritance is not overridable. Instead, it works *monotonically*, by accumulation. In our example, the class `WebNewsArticle` is said to be a subclass of `PublicWebDocument` in Figure 4, so the two types would be considered consistent for nearly all applications. In general, if $\mathcal{E}$rgoAI derives different types for an attribute of an object, the types are not taken to be contradictory. This viewpoint is similar to that of formal ontologies in which

[14] Throughout this paper we adopt the convention that identifiers for classes begin with an upper-case letter.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

two concepts are not taken to be disjoint unless a relevant exclusion axiom causes them to be. In ℰrgoAI, similar constraints can be enforced either by type checking or by integrity constraints (see Section 7). The paper Yang and Kifer (2003) gives rigorous treatment to the semantics of inheritance in ℰrgoAI.

- *Monotonic behavioral inheritance.* While behavioral inheritance is defeasible by default in ℰrgoAI, there are cases when monotonic inheritance is preferred. In our example the people described by the knowledge base are assumed to speak English, but they may speak other languages as well. ℰrgoAI allows behavioral inheritance to be set as either monotonic or non-monotonic at the module level. Figures 3 and 4 and the attribute `language` illustrate this point. The default strategy for behavioral inheritance is non-monotonic and thus the language (or languages) stated at Bunky Muntner's object level in Figure 3 overrides the inheritance of `language` from class `Person` specified in Figure 4. If, however, the module were told to use the monotonic semantics for behavioral inheritance (say, by inserting the directive `:- setsemantics{inheritance=monotonic}` then `language->English` would be inherited *monotonically* from class `Person:` that isit would be added to the statement `language->Spanish` specified directly inside the object for Bunky Muntner. Thus a query such as

```
?- person('Bunky Muntner')[language->?Language].
```

would unify `?Language` with both `English` and `Spanish`, as desired.

3.2 Attributes defined via rules

Attributes can be defined by ℰrgoAI rules, facts, or both. As an example we show how ℰrgoAI can represent and maintain contemporary knowledge sources such as statement embeddings used by language models (LMs).

Both LLMs and small LMs such as Phi-3 (2024) and Google's Universal Sentence Encoder Cer *et al.* (2018) are usually written in Python and C^{++} and so are easily accessible by ℰrgoAI through the Janus interface Swift and Andersen (2023). For instance, suppose embeddings were obtained using a version of Universal Sentence Encoder, whose Python API loads a model, and then calls it with different textual inputs. The model object is created by the goal.

`Py_func(tensorflow_hub,load(`$<$*model_url*$>$`),?PyObj)`

The predicate `Py_func/3` allows ℰrgoAI to call a Python function, in this case

`tensorflow_hub:load(`$<$*model_url*$>$`).`

The return of this call unifies the ℰrgoAI variable `?PyObj` with a Python object reference of the form `pyObj(ObjectReference)`. After asserting the object reference as a `currentModel/1` fact, a rule for `Claim` objects can be written as

```
?Obj:claim[embedding->?Vector]:-
    currentModel(?Model),
    ?Obj[sourceText->?Text],
    Py_dot(Model,embed(Text),?Vector).
```

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

where `?Vector` will be instantiated to a list of floating point numbers. Unlike this explanatory example, in practice it is usually best to keep information such as embedding vectors as attributes of Python objects, which can be accessed and manipulated through Janus in much the same manner as $\mathcal{E}$rgoAI objects.

3.3 Type checking

A type in $\mathcal{E}$rgoAI can be any class in the classification hierarchy. Remembering that the class membership (the `:/2` relationship), the class hierarchy (the `::/2` relationship), and the properties (the `->/2` relationships) can be defined via logic rules, it is no wonder that $\mathcal{E}$rgoAI types are powerful and expressive but also that fully static type checking is not possible. *Dynamic* type checking *is* possible. While dynamic type checking cannot be guaranteed to terminate and may be slow in complex cases, it is often a highly useful aid to program development. To perform dynamic type checks, one can easily express different type *violation* as queries that return type violations as *counterexamples* to type correctness.

Although the semantics needed for signatures is sometimes program-specific, many type inconsistencies are common, including (1) that an object has an attribute value inconsistent with the inferred type for that value; (2) an object has an attribute that is not in any of its inherited signatures; or (3) the number of values for a given object and attribute are not consistent with cardinality constraints declared in an inherited signature. Type checking can be particularly important when complex objects are inserted in a dynamic setting, as will be discussed in Section 8.

Here is an example of type violations expressed as a query that identifies attributes and values for instances that violate properties 1 and 2 above.

```
type_error(?Obj,?Attr,?R,?D) :-
       %% Values that violate typing
       ?Obj[?Attr->?R], ?Obj[?Attr=>?D], \naf ?R:?D
       ;
       %% Defined methods that do not have type information
      ?Obj[?Attr->?R], \naf ?Obj[?Attr=>?_D].

?- type_error(?Object,?Attr,?Result,?Class).
```

To save the user the trouble of writing type constraints for common cases of type violations, $\mathcal{E}$rgoAI provides the `\typecheck` module that supports the above type checkers so that the user can simply query:

```
Type[check(person('Bunky Muntner')[?Meth->?],?Result)]@\typecheck.
```

Details can be found in the $\mathcal{E}$rgoAI user's manual.

3.4 $\mathcal{E}$rgoAI frame compilation

The compilation of $\mathcal{E}$rgoAI frames into Prolog predicates follows the general design of Kifer *et al.* (1995) and consists of two parts: (1) frames are translated into Prologs; and

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

(2) *closure rules* are added to support the object-oriented semantics of frames. In this section, we describe a few aspects of ℰrgoAI compilation to give the reader an idea of the compilation process.

Both instance and class frames are translated into Prolog in a similar manner, so we take as an example the following instance frame, which is to be added to ℰrgoAI's knowledge base. The frame is *complex* because it has multiple attribute–value pairs, because its identifier includes an inheritance predicate (`: /2`), and because it has a set-valued attribute.

```
Mary : employee[age->29, salary(1998)->100000, kids->{Tim,Leo}].
```

As described in Kifer *et al.* (1995), frame compilation begins with translating *complex* frames (like the above) into a conjunction of *atomic* frames. As part of this process, attribute values that are sets are split into a conjunction of singletons, and the membership statement for `Mary` is factored out of the frame:

```
Mary : employee,Mary[age->29],Mary[salary(1998)->100000],
                    Mary[kids->Tim],Mary[kids->Leo].
```

These atomic frames are then translated to Prolog facts that are asserted to the XSB database as *tries* Ramakrishnan *et al.* (1999).

```
isa(Mary,employee).
mvd(Mary,age,29).                    mvd(Mary,salary(1998),100000).
mvd(Mary,kids,Tim).                  mvd(Mary,kids,Leo).
```

The *wrapper predicate* `mvd/3` (which stands for multi-valued data) is used for the translation of attributes with behavioral inheritance. Query frames must undergo the same translation as program frames for resolution to work properly. For instance the query

```
Mary:?Class[kids->?Kid].
```

would be translated to

```
isa(Mary,?Class),mvd(Mary,kids,?Kid).
```

yielding two solutions, one with {`?Class=employee,?Kid=Leo`} and another with {`?Class=employee,?Kid=Tim`}. ℰrgoAI's compiler mangles the names of the wrapper predicates to include the module of the file or query.

ℰrgoAI rule frames must be encoded in a more complex manner than fact frames and require generation of *closure rules* to:

- Compute the transitive closure of `:: /2` in a manner that supports runtime warnings about cycles in the subclass hierarchy.
- Compute the closure of `: /2` with respect to `:: /2`, that is if `X : C` and `C :: D` then `X : D`.
- Perform inheritance. In particular, monotonic inheritance must detect and propagate multiple type values while non-monotonic inheritance must detect and override object value conflicts.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

- Provide an infrastructure to treat undefined methods as exceptions. Unlike Prolog, undefined predicates in $\mathcal{E}$rgoAI do not throw errors by default. However when checks for undefinedness are turned on, code to raise appropriate errors is injected into the closure rules.

The above closure rules are specified statically in special files that get included in the compiled Prolog code after the appropriate preprocessing that makes the files compliant with $\mathcal{E}$rgoAI's dynamic module system.

4 HiLog

The previous section showed how complex frames, called *frame formulas*, can be constructed and used, but the examples also contained logical terms such as the object id `claim(13344)` and the occurrence of `py_func/3` in the body of a rule. $\mathcal{E}$rgoAI fully supports HiLog Chen *et al.* (1993) which gives a higher-order syntax to Prolog-style resolution. The idea behind HiLog can be explained easily. For instance, the HiLog fact

```
p(fact(7))(a,f(b,b),[3,?X])
```

is schematically translated to Prolog as:

```
apply(apply(p,fact,7),a,apply(f,b,b,[3,?X])).
```

where every non-list compound term T is translated into a term with functor `apply/n` and T as its first argument. By similarly translating a query such as

```
?- p(?X)(a,f(b,b),[3,foo])
```

to

```
apply(apply(p,?X),a,apply(f(b,b),[3,foo])
```

the query can be resolved against the fact with either tabled or non-tabled Prolog-style resolution. To be more specific, a HiLog term is defined as follows and this semantics is equivalent to the model-theoretic view of HiLog Chen *et al.* (1993).

Definition 1.
A HiLog atomic constant *is an atom. A* base HiLog term *is a HiLog atomic constant, a list, a variable, or an arithmetic expression.*

A general HiLog term *is a base HiLog term, or has the form* $T(\overline{Args})$ *where* T *is a general HiLog term, and* $\overline{Args}$ *is an argument list of general HiLog terms.*

HiLog is sometimes said to have a higher-order syntax but a first-order semantics. Specifically HiLog semantics is based on a translation into Prolog.

Definition 2.
The HiLog transform tr_H *is defined as follows.*

If $Term$ *is a base HiLog term, then* $tr_H(Term) = Term$. *Otherwise if* $Term = Term_1(\overline{Args})$ *then*

$$tr_H(Term) = apply(tr_H(Term_1), tr_H(\overline{Args}))$$

where $tr_H(t\overline{Args})$ *simply applies* tr_H *to each argument in the sequence.*

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

It can easily be seen that by applying tr_H to a HiLog term produces a Prolog term. As a result, if tr_H is applied both to HiLog programs and goals in a program $\mathcal{P}$, the model of $\mathcal{P}$ coincides with the model of its translation. Operationally, we refer informally to resolution of translated goals as *HiLog resolution*.

Since variables can appear nearly anywhere in HiLog terms, HiLog is useful for exploring the structure of knowledge bases. Another use case is traditional higher-order programming. Suppose we need to transitively close many different binary relations, like `flight/2`, `edge/2`, `parent/2`. Some of these needs may arise in the future and not be known at present. Instead of writing the usual 2-rule definition of the transitive closure in each such case, we could write just one pair of rules for all current and future needs:

```
closure(?P)(?F,?T) :- ?P(?F,?T).
closure(?P)(?F,?T) :- closure(?P)(?F,?Mid),?P(?Mid,?T),
```

Note the implicit reliance on tabling for the second, left-recursive rule. Now, to compute the transitive closure of `flight/2` and of `edge/2` we can simply pose these two queries:

```
?- closure(flight)(?F,?T).
?- closure(edge)(?F,?T).
```

Avoiding name clashes: In systems that are implemented via a translation to a target system, name clashes with other components of the target system become a concern. This consideration applies not only to the HiLog *wrapper predicate* `apply` but also to frame wrappers `mvd/3`, `isa/2`, etc. (cf. Section 3.4). Such unintended interactions are usually hard to detect and debug. $\mathcal{E}$rgoAI addresses this problem by *salting* (also called mangling) the wrapper predicates with strings of characters that are unlikely to occur under normal circumstances. Thus, instead of `apply`, `mvd`, etc., the salted versions of these wrappers are used: '*salt*`^apply'`, '*salt*`^mvd'`, '*salt*`^isa'`, etc., where *salt* denotes the string that includes rare characters such as the quotes, $, and various others. For instance, $\mathcal{E}$rgoAI uses `'_$_$_ergo'` to salt the wrapper predicates used in HiLog/F-logic-to-Prolog translations.

5 The module system of $\mathcal{E}$rgoAI

In HiLog and F-logic, the number of unique predicates used for the Prolog translations is small. As a result, it is easy to write a goal that backtracks through all clauses in the program, a behavior that can lead to inefficiency. The salting technique of the previous section prevents unintended resolutions between $\mathcal{E}$rgoAI's wrapper predicates and similarly named predicates from other XSB programs that may be used in conjunction with $\mathcal{E}$rgoAI in the same XSB process. Without salting there could be name clashes between the `mvd` and `apply` wrappers discussed in previous sections and `mvd` or `apply` used in other XSB programs (if XSB predicates are in the default `usermod` module). But there is another problem: the separation of namespaces among the wrappers *within* the $\mathcal{E}$rgoAI system. Such separation is necessary to enable component-wise construction of large knowledge bases and to reduce the chances of inadvertent interactions. As in most languages, $\mathcal{E}$rgoAI's module system addresses these and similar issues.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

$\mathcal{E}$rgoAI's module system was designed to be dynamic and to support a number of other unique features. Modules can be created "on the fly" at run time, and rules and data can be loaded into them, inserted, or deleted at run time as well. Calls from one module to another may also be constructed at run time. At the same time, any module can prevent unwanted calls by encapsulating itself. Encapsulation can happen at compile time or dynamically, via the executable `export/1` directive (e.g., `export{?[abc ->?], pqr(?)}` or even `export{?[abc ->?], pqr(?)}>> module1,module2`). Furthermore, modules need not be instantiated within a goal; for instance one can query the modules and argument values for which `p/1` is true:

```
?- p(?X)@?M.
```

Some of the above features exist in various logic programming systems, but others (variables over modules, support for non-predicate-based languages like HiLog and F-logic) are unique to $\mathcal{E}$rgoAI.

The implementation of $\mathcal{E}$rgoAI modules elaborates on the idea of salting the wrappers such as `apply/n` and `mvd/3`. The goal is that when wrappers] used in *different* modules are translated using *different* salted wrappers. For example, the statements `John[age->11]` and `Mary[age->12]` that appear in the *same* module would be related to each other because they would be encoded using the same set of salted wrappers: like '*salt*`^mvd'`, '*salt*`^isa'`, or '*salt*`^apply'`. In contrast, these same statements in *different* modules would be encoded using *different* salted wrappers.

5.1 Module-aware salting

Thus, the basic idea underlying the implementation of $\mathcal{E}$rgoAI modules is that in the end, all HiLog and F-logic wrapper predicates are translated with respect to a particular module, and the module symbol is used only in the outermost functor of the translated wrapper predicates. This restriction is made because ultimately the system determines the truth values of the instances of the wrapper predicates only. For instance, the module-specific version of HiLog's `apply/n` is '*salt*`^`*modname*`^apply'` $/\mathrm{n}+1$. Functors at inner levels become arguments to the outer wrappers – after the HiLog and F-logic transforms, inner functors are no longer treated as predicates, have no truth values, and their salting does not involve modules. As a result, the arguments conceptually play the role of shared object identifiers about which logical assertions (facts and rules) are being made. Assertions made in one module about a set of objects do not interact with the assertions about that same set of objects in another module unless one module explicitly queries the other.

For example: suppose a database table named `p/1` exists both in module `foo` and module `bar`. A priori we do not know of any relationship between these two tables so their contents should be considered to be unrelated. However, if we learn that module `foo` has a rule of the form

```
p(?X) :- p(?X)@bar.
```

then the version of `p/1` in `foo` must necessarily have all the tuples that the version of `p/1` in `bar` has.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

Admittedly, salting only the outer functor of a term means that we distinguish between `p(?X)` that appears as a call and `p(?X)` that appears as an argument, which makes meta-programming noticeably less elegant. This trade-off is well known and analogous to choosing between a Prolog with a predicate-based module system and a Prolog with a functor-based module system (e.g., Ciao Prolog vs. XSB).

Because assertions are module-specific, queries are issued from a particular module, although they may involve querying another module. For instance,

```
?- p(?X)(a,f(b),[3,?X]), q(?X)@bar.
```

translates into a query of the form

```
?- 'salt^modname^apply'('salt^apply'(p,_X),a,'salt^apply'(f,b),[3,_X]),
    'salt^bar^apply'(q,_X).
```

Similarly, the wrapper predicates *'salt^modname^*`mvd'`, *'salt^modname^*`isa'`, etc. are used to separate translations of the frame statements into different modules. For rules, the translation is similar:

```
hd(?X) :- ?Y:?X@bar, foo::?Y.
```

becomes

```
salt^modname^apply'(hd,_X) :-
    'salt^bar^isa'(_Y,_X), 'salt^modname^sub'(foo,_Y).
```

Note that after the compilation of the above query, the module-part of the first outermost wrapper predicate, *salt^modname^*`apply'/4`, remains generic, that is, it is not replaced with a real module name. Similarly, in the above rule, the module parts of the wrappers *salt^modname^*`apply'/2` and *'salt^modname^*`sub'/2` remain generic. As may be guessed, this happens because the above goal `p(?X)(...)` does not have any module reference (@*mod*); likewise, neither `hd(...)` nor `foo::?Y` in the above rule has such references. So, what is the module in those cases and how would the compiler obtain that module's name? In ErgoAI, the answer is that the module in question is the "*current*" module, a concept closely tied to the dynamic manner in which rules and modules are created and associated with each other during the course of a computation.

5.2 Loading versus adding

Normally the user creates a file that includes facts, rules, and queries. Then the user loads or adds that file to a module (the module may or may not exist at the time), which causes the file to be compiled *for that module* and the queries executed. The user might pose additional queries, load, or add additional files into new or existing modules, or perform other operations.

For concreteness, suppose we have a file called `myProgram.ergo` that contains the query and the rule just discussed plus some data facts (i.e., bodiless rules). We can load the file into some explicit module, for example `mainRules`, or into the *current* module:

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

```
?- [myProgram >> mainRules].
?- [myProgram].
```

The first form will create the empty module `mainRules`, if it did not exist before, compile `myProgram.ergo` for the module `mainRules` causing compiled salted Prolog rules to appear in XSB's memory. (If the module existed prior to the command, it is *emptied out* of its old contents first.) In the second version of the load command, `myProgram.ergo` will be compiled for the *current* module but the rest is the same as we just explained.

The current module dynamically changes during an $\mathcal{E}$rgoAI session. When $\mathcal{E}$rgoAI starts up, it automatically creates the module `main` which is the first current module. If either of the above load commands executes next, it executes with `main` as the current module. During the compilation of `myProgram.ergo` using the first option, the current module will be `mainRules` rather than `main`. With the second version of the load command, the current module will not change, of course, and will still be `main`. If `myProgram.ergo` includes another loading command, like `?- [myStuff >> miscRules]`, then when this command is executed, `myStuff.ergo` will be compiled with `miscRules` as the current module.

Loading wipes out any existing contents of the target module, so loading two different sets of rules into the same module is rarely meaningful unless the first set of rules and data has become obsolete. Such scenarios arise when dealing with streaming data, but are unlikely in cases where working knowledge of an application evolves. For such applications, $\mathcal{E}$rgoAI provides the *add* command, for example,

```
?- [+myAddlStuff >> mainRules].
?- [+myAddlStuff].
```

Using the add command resembles loading except that it does not erase beforehand the contents of the module (`mainRules` in the first form, the current module if the second form). The overall result of $\mathcal{E}$rgoAI's dynamic creation of modules, loading, and adding is that any given file can either be loaded into any module or added at any time.[15]

5.3 Compiling dynamic module references

Until now, we remained silent about how the placeholder for the current module, *modname*, gets replaced with the actual name of that module within the context of a given execution. This is accomplished with the help of the GPP preprocessor[16] a core program used by XSB as well. GPP is similar to the CPP preprocessor used by C and C++, but includes features needed by $\mathcal{E}$rgoAI that CPP does not support such as the ability to change syntax on the fly and to perform macro evaluations inside quoted strings.

Because code files may be dynamically loaded or added into modules, $\mathcal{E}$rgoAI does not compile into the XSB Prolog syntax directly but rather into a syntax where HiLog and frame wrappers are enveloped with suitable GPP macros. For example, instead of compiling the `apply/n` wrapper into `'salt^bar^apply'(q,_X)` and

[15] In Prolog, files may be conditionally loaded into a module, but this approach is somewhat more restrictive than in $\mathcal{E}$rgoAI.
[16] https://github.com/logological/gpp

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

`'salt^modname^apply'('salt^apply'(p,_X),a,'salt^apply'(f,b),[3,_X])`
directly, we could use these GPP macros:

```
#mode save
#mode nostring "\!#'"
   #define  PREFIX                        _$_$_ergo
   #define  SALT_WRAP(BaseWrap)           'PREFIX^BaseWrap'
   #define  MODSALT_WRAP(Mod,BaseWrap)    'PREFIX^Mod^BaseWrap'
   #define  CURRMOD_WRAP(BaseWrap)        'PREFIX^CRNT_MODULE^BaseWrap'
#mode restore
```

and compile our HiLog terms into these more structured and compact expressions:

```
MODSALT_WRAP(bar,apply)(SALT_WRAP(apply)(q,_X))
CURRMOD_WRAP(apply)(SALT_WRAP(apply)(p,_X),a,SALT_WRAP(apply)(f,b),[3,_X])
```

There are several interesting things to note about these macros. First, the last three macro definitions refer to other macros such as `#PREFIX`, and these referenced macros appear inside quoted strings.[17] The `#mode save/restore` commands above order GPP to temporarily stop interpreting single quotes as string delimiters, which enables GPP to compute all those macros as we intended. Second, notice that `CRNT_MODULE` is undefined as a macro, but it is clear from the name that we somehow want it to evaluate to the current module name (which recall, changes depending on the context where the reference to the current module appears). In our example, the current module can be `main` or `mainRules` and this becomes known only at the time of loading. Fortunately, GPP can be forked from its parent process and can take a directive to add a new macro

```
#define CRNT_MODULE   module-being-loaded
```

where *module-being-loaded* is the module name known to the loader. In other words, the problem of passing the name of the current module is solved because we can pass that name to GPP under the guise of the `CRNT_MODULE` macro.

As a final point, ℰrgoAI's module system also extends so that Prolog modules are called in a manner similar to ℰrgoAI modules. For instance, the ℰrgoAI goal that invokes the Prolog predicate `member/2`, which lives in the XSB module `basics`, is:

```
member(?X,?List)@\prolog(basics).
```

This is implemented simply by compiling `member/2` without any salting whatsoever. Since the core of XSB is written in C, C routines appear as regular Prolog calsl, so in ℰrgoAI they also can be accessed the same way, via the `@\prolog` idiom. In principle, this idiom could also be extended to other languages so that, for example, Python modules would be accessible via `@\python` or `@\janus`, if Janus is used.

[17] As far as we know, the CPP preprocessor in C/C++ cannot be forced to evaluate such macros, but XSB's GPP can.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

6 Rules and defeasibility

As mentioned in Section 2, $\mathcal{E}$rgoAI's default negation (\naf) is based on WFS. In Section 3 we also described how $\mathcal{E}$rgoAI supports both defeasible behavioral inheritance and monotonic inheritance of signatures (i.e., types). At this point we turn our attention to $\mathcal{E}$rgoAI's support for explicit negation and several versions of defeasible reasoning in the LPDA style Grosof (1997); Wan *et al.* (2009), each defined by its own *argumentation theory*. In fact, any properly defined argumentation theory for $\mathcal{E}$rgoAI can be enabled in a user module, and different such theories can be active at the same time as long as they are in different modules.

An argumentation theory in $\mathcal{E}$rgoAI acts behind the scenes, providing the ability to extend the base SLGR(A) resolution strategy of the system in a well-defined, declarative way. $\mathcal{E}$rgoAI's default argumentation theory is *Generalized Courteous Logic Programming (GCLP)* introduced in Grosof (1997); Wan *et al.* (2009), which $\mathcal{E}$rgoAI further extends with *cancellation* along with other refinements developed in the 15+ years since those publications. A typical argumentation theory in $\mathcal{E}$rgoAI is visible to a user through three predicates: \opposes, \overrides, and \cancel. It is through this interface that a user can control how defeasibility is handled by the reasoning system. We first describe the overall effect of defeasibility on $\mathcal{E}$rgoAI's inference through an example, next examine the above predicates for defeasibility control, and then we discuss their semantics.

Figure 5 shows a fragment of a set of beliefs about claims that we assume are loaded into the module `Belief`. The directive `:- use_argumentation_theory` at the top ensures that $\mathcal{E}$rgoAI's default argumentation theory will be activated when this fragment of the knowledge base is loaded into a module. The LPDA support enhances the syntax of $\mathcal{E}$rgoAI with the ability to optionally attach *rule descriptors* to each rule. In our example they are colored in shades of green and appear as `@!{...}` and `@{...}` constructs. The `@!{...}` portion of the descriptor assigns a unique *rule id* and provides meta-data about the rule; if no rule Id is explicitly associated with a rule by the author, one is created automatically. The `@{...}` portion of the rule descriptor, is a *rule tag*, which makes the rule *defeasible* (as opposed to *strict*, i.e., non-defeasible). These tags are used heavily in defeasible reasoning.[18]

Rule descriptors provide meta-data for rules by means of frames that can be used much in the same way as other $\mathcal{E}$rgoAI frames. In Figure 5, the rule descriptors have attributes such as `author` and `comment` that resemble those of other rule systems. However, by querying rule descriptor frames using the construct `@!{...}`, a rule can be linked into a classification hierarchy in which, for example,

```
defaultPotencyRule::PotencyRule.        PotencyRule::ViralRule.
```

One easy way to do this is by adding the following rule to the `Belief` module:

```
?RuleId:?Class :- @!{?RuleId[isa->?Class]}.
```

[18] Rule tags can alternately be written by placing the attribute–value pair `tag->Potency(...)` but in many cases the short form `@{`*SomeTag*`}` of the descriptor is more convenient than the general form `@!{`*SomeId*`[...,tag->`*SomeTag*`,...]}`

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

```
:- use_argumentation_theory.  // Use the default argumentation theory
// Defeasible rules with rule descriptors
@!{pr1[author->Swift,isa->PotencyRule]}
@{Potency(High,?Per,?V)}
?Per[believes->hasPotency(?V,High)] :-
        ?V:Virus,
        ?Per[residesIn->?Loc],
        ?_Claim:Claim[
           claimer->?Per,
           statement->${?Loc[shouldRequire->SchoolClosure[purpose->reduceXmit(?V)]]}].
@!{dr1[author->Kifer,isa->DefaultPotencyRule]}
@{Potency(Low,?Per,?V)}
?Per[believes->hasPotency(?V,Low)]:- ?Per:Person, ?V:Virus.
// Defeasibility control statements
\opposes(Potency(?L1,?Per,?V),?_AnyGoal1, Potency(?L2,?Per,?V),?_AnyGoal2) :-
        ?L1 != ?L2, ?V:Virus, ?Per:Person.
\overrides(Potency(High,?_Per,?_V), Potency(Low,?_Per,?_V)).
\cancel(Potency(?,?,?)):- \neg Pandemic.  // Cancel Potency rules if no pandemic
// Some other (strict) rules
?Per[residesIn->?GPE]:- ?Per[residesIn->?Loc], ?Loc[city->?GPE].
?Per[residesIn->?GPE]:- ?Per[residesIn->?Loc], ?Loc[state->?GPE].
// Base Facts
ECM:Virus.
person('Bunky Muntner'):Person
 [ age->63,
   residesIn->zip(20016)[city->Washington,state->DC],
   spouse->'Ginger Muntner'].
claim(12345):Claim
 [ medium-> Instagram,
   topic->'Government Containment Actions: School Closures',
   location->'Washington DC',
   statement->${zip(20016)[shouldRequire->SchoolClosure[purpose->reduceXmit(ECM)]]},
   claimer->person('Bunky Muntner')].
// The query: what do the people believe should be done?
?-  ?Per:Person[believes->hasPotency(?V,?Strength)].
```

Fig. 5. A fragment of a belief logic.

Rule tag descriptors (`@{...}`) are used by ℰrgoAI's argumentation theories. In general, a tag descriptor can contain any valid frame or HiLog term. The rule with the tag `Potency(High,?Per,?V)` says that a person believes the potency of a virus is `High` if the person has made a public claim that their local schools should be closed. The rule with the tag `Potency(Low,?Per,?V)` serves as a default rule.

The syntax ${...} – seen for instance in the body of the low potency rule – *reifies* logical formulas such as frames into objects that can be used as values of attributes or as arguments of HiLog terms and predicates. In this example, reification allows the representation of the beliefs of different voters about school closing.

The three *defeasibility control predicates* in the middle of Figure 5 extend the GCLP argumentation theory to be domain-specific. `\opposes` /2, provides conditions for (resolvable) contradiction beyond the implicit contradiction that always exists between an atom A and its explicit negation `\neg` A. In this example the opposition statement enforces the semantic constraint that a rational person should not believe that the same virus has different overall potencies. In addition to marking rules as defeasible, the rule tags

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

of Figure 5 are also used for prioritization via \overrides. In our case, the \overrides /2 fact ensures that, for a given person and virus, any other potency rule overrides the Low potency rule. Note that by using the tags, argumentation predicates can be coded to apply to precise sets of rules or derived heads. Finally, the \cancel rule ensures that all potency rules are disregarded by the belief logic unless there is an exogenously defined state of a pandemic.

At this point, it is instructive to try to understand at an intuitive level what defeasibility of the first two rules in Figure 5 has given us. If we ask the query shown as the last statement in that figure:

```
?- ?Per:Person[believes->hasPotency(?V,?Strength)].
```

we get:

```
?Per = person('Bunky Muntner')  ?V = ECM   ?Strength = High
```

which seems an answer we intuitively expect. However, if we focus on just the rules and the base facts, while ignoring the defeasibility control predicates, then our knowledge base appears to imply two beliefs:

```
person('Bunky Muntner')[believes-> hasPotency(ECM,High)]
person('Bunky Muntner')[believes-> hasPotency(ECM,Low)]
```

where the first fact is derived via the first rule in the figure (with Id pr1) and the second fact by the second rule with Id dr1. On the other hand since person('Bunky Muntner'): Person and ECM: Virus,

```
opposes(Potency(High,person('Bunky Muntner'),ECM),
        Potency(Low,person('Bunky Muntner'),ECM))
```

is also derived, so that the above two belief statements cannot both be true. The \overrides statement says that the derivations made via rule pr1 (high potency) are more important than derivations made by rule dr1 (low potency-low). As a result, the answer of the belief in low potency above gets discarded and our knowledge base returns only the first (high-potency) answer.

As an experiment, one could try to comment out the defeasibility control predicates and pose the same query again. Now we get *two* answers, consistent with what the normal derivation in $\mathcal{E}$rgoAI yields (sans defeasibility).

```
?Per = person('Bunky Muntner')  ?V = ECM   ?Strength = High
?Per = person('Bunky Muntner')  ?V = ECM   ?Strength = Low
```

So, clearly defeasibility control predicates have an effect. This effect is the result of $\mathcal{E}$rgoAI augmenting the knowledge base with an *argumentation theory* that specifies what to do if one kind of rule overrides the other, one kind of derived face conflicts with another, etc. During normal operation, the argumentation theory is applied behind the scenes and is completely transparent to the user.[19]

[19] On the other hand, the actions of opposition, overriding, and cancellation can all be examined when debugging.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

6.1 The GCLP argumentation theory

Pseudo-code for *GCLP*, ℰrgoAI's default argumentation theory is shown in Figure 6. The actual code in ℰrgoAI is more complex due to optimizations and debugging support.[20] That code also exports a number of query points that let the user investigate why certain rules were defeated, refuted, or rebutted; why there is a conflict between some rules; etc. As mentioned, ℰrgoAI supports several argumentation theories other than GCLP that differ in how rules are defeated, and in the generality of opposition. For instance, some argumentation theories allow `\opposes` /2 rules to specify when a set of three or more different rules must succeed for opposition to happen.

Regardless of the argumentation theory, a defeasible rule:

$$@\{RuleTag\}\ Head\ \texttt{:-}\ Body$$

is translated to

$$@\{RuleTag\}\ Head\ \texttt{:-}\ Body,\ \texttt{\textbackslash naf}\ defeated(RuleTag,Head)@AT.$$

where AT is the ℰrgoAI module that contains the desired argumentation theory. Observe that this simple cross-module calling scheme enables different ℰrgoAI modules to use different argumentation theories at the same time. A defeasible rule with variable bindings for which *Body* succeeds is called a *candidate.*

It can be seen from Figure 6 that a given rule, say R_t with tag *Tag* (denoted R_t/Tag, is defeated if it is refuted or rebutted by some non-defeated rule, or if R_t/Tag is disqualified. Rebuttal requires that an opposing rule, say R_o, with tag *OppTag* (denoted $R_o/OppTag$) succeeds as a candidate, and that $R_o/OppTag$ is neither canceled nor refuted. Refutation, on the other hand, also requires an appropriate `\overrides` /2 statement to be present. Thus in GCLP if two undefeated rules rebut each other but neither is overridden, both fail. Otherwise the `overrides` statement causes either one rule to succeed and the other to fail, or both to succeed with truth value u in the case of mutual overriding. Rule sets like that in Figure 6, when optimized for performance, implement ℰrgoAI's argumentation theories.

We should note that the literature is rife with flavors of defeasible logic Prakken (1993); Nute (1994); Gelfond and Son (1998); Brewka and Eiter (2000); Sakama and Inoue (2000); Wang *et al.* (2000); Dung and Tran (2001); Zhang *et al.* (2001); Delgrande *et al.* (2003); Eiter *et al.* (2003), each slightly different from the next. It is argued in Wan *et al.* (2009, 2014) that argumentation theories, like those in ℰrgoAI, can simulate and approximate a number of other approaches, and the dozen or so argumentation theories supplied with ℰrgoAI testifies to the power and flexibility of that approach.

6.2 Argumentation and paraconsistency in ℰrgoAI

The following example highlights a property of ℰrgoAI's GCLP argumentation theory: the failure of rebutted rules.

[20] GCLP in ℰrgoAI also allows strict rules to override defeasible rules in cases of opposition.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

```
defeats(?Tg1,?Hd1.?Tg1,?Tg2):- refutes(?Tg2,?Hd2,?Tg1,?Hd1).
defeats(?Tg1,?Hd1.?Tg1,?Tg2):- rebuts(?Tg2,?Hd2,?Tg1,?Hd1).

defeated(?Tg1,?Hd1):- refutes(?Tg2,?Hd2,?Tg1,?Hd1).
defeated(?Tg1,?Hd1):- rebuts(?Tg2,?Hd2,?Tg1,?Hd1).
defeated(?Tg1,?Hd1):- disqualified(?Tg1,?Hd1).

rebutted(?Tg2,?Hd2):- rebuts(?,?,?Tg2,?Hd2).

rebuts(?Tg1,?Hd1,?Tg2,?Hd2) :-
      conflicts(?Tg1,?Hd1,?Tg2,?Hd1),
      \naf refuted(?Tg1,?Hd1).

refuted(?Tg2,?Hd2):- refutes(?,?,?Tg2,?Hd2).

refutes(?T1,?Hd1,?Tg2,?Hd2) :-
      conflicts(?Tg1,?Hd1,?Tg2,?Hd2),
      \overrides (?Tg1,?Hd1,?Tg2,?Hd2),
      \naf refuted(?Tg1,?Hd1).

conflicts(?Tg1,?Hd1,?Tg2,?Hd2) :-
      \opposes(?Tg1,?Hd1,?Tg2,?Hd1),
      candidate(?Tg1,?Hd1),candidate(?Tg2,?Hd2).
      \naf \cancel(?Tg1,?Hd1), \naf \cancel(?Tg2,?Hd2).

candidate(?Tg,?Hd) :- getBody(?Tg,?Hd,?Body), call(?Body).

disqualified(?Tg2,?Hd2):- transitively_defeats(?Tg1.Hd1,?Tg2,?Hd2),
                          \naf rebutted(?Tg1,?Hd1).
disqualified(?Tg,?Hd):- \cancel(?Tg,?Hd).
```

Fig. 6. Simplified pseudo-code for GCLP, $\mathcal{E}$rgoAI's default argumentation theory.

Example 1 (Explicit Negation in GCLP).
Given the preceding discussion, it is useful to consider the semantics that GCLP assigns to the opposition of defeasible atoms `A` *and* `\neg A` *in Program* P_{eneg}

$\mathbf{P_{eneg}}$:

```
:- use_argumentation_theory
 @{tag1} A.       @{tag2} \negA.
```

Here A and `\neg A` *are both facts and thus candidates. Since they have tags, they are defeasible. GCLP and many other LPDA-style argumentation theories make an implicit assumption that* `A` *and* `\neg A` *always oppose each other. Due to the use of GCLP in Program* P_{eneg} *the defeasibility transformation ensures that every defeasible rule R is made to guard against being defeated (i.e., the literal* `\naf` defeated(RuleTag$_R$, Head$_R$)@AT *is added to the body of R). Therefore, by the* `refutes/4` *rule in Figure 6, they conflict. By the* `rebuts/4` *rule in Figure 6 we see that* `A` *and* `\neg A` *rebut each other. Since neither of these facts is refuted, they are both defeated, so that both* `A` *and* `\neg A` *are false.*

GCLP's \treatment of explicit negation is the same as most of $\mathcal{E}$rgoAI's argumentation theories, but it differs from approaches like WFSX Alferes *et al.* (1994) in which both `A` and `\neg A` would have the truth value *undefined*. The fact that GCLP sets their truth value to *false* makes it arguably more useful for *paraconsistent* reasoning based on frame

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

logic with argumentation. If the truth value of opposing literals were set to *undefined* then the truth values of literals that depend on these opposing literals would also be set to *undefined*.

Although some logics use a four-valued lattice to distinguish unknown from incompatible truth values, like those in P_{eneg}, handling inconsistency purely within WFS requires mapping inconsistency to an existing truth value: true, false, or undefined. Viewed in this way, WFSX conflates the inconsistency of a literal with a literal's being undefined/unknown (u) while GCLP conflates it with *false*. In fact, ℰrgoAI users have suggested a way to support a four-valued truth lattice semantics based on an argumentation theory that acts like GCLP *except* that it has no concept of rebuttal. This argumentation theory, `RefuteCLP`, is included with ℰrgoAI. In `RefuteCLP`, if one rule refutes another, any inferences drawn by the second rule are defeated. However, if two rules rebut (but do not override) each other then neither inference is defeated and it is possible to infer that, in a pair of opposing facts, both are true. Therefore, if P_{eneg} in Example 1 used RefuteCLP instead of GCLP, both `A` and `\neg A` would have been true. ℰrgoAI does not provide direct support for the fourth truth value, *inconsistent*, but user-built rulebases can treat such situations as *localized inconsistency*. Localized inconsistency is similar to paraconsistency in Annotated Logic Blair and Subrahmanian (1989); Kifer and Lozinskii (1992), and sophisticated users can employ meta-interpreters or similar techniques to detect and take advantage of paraconsistency in ℰrgoAI knowledge bases.

7 Flexible search strategies for unsafe subgoals

As mentioned in Section 2 ℰrgoAI does not require a program to be ground to be evaluated, a major factor in its scalability. However, this freedom can give rise to *unsafe* queries that do not satisfy a required instantiation pattern. Given ℰrgoAI's reliance on well-founded negation within non-monotonic inheritance and argumentation, unsafe negation must be addressed. In addition, ℰrgoAI's use of external knowledge may also give rise to unsafe queries that must be addressed.

ℰrgoAI's approach to unsafe queries uses *instantiation delay* together with the WFS u truth value. Instantiation delay is different from the *SLGR(A)* delay presented in Section 2, although the two mechanisms do bear some resemblance. As discussed, *SLGR(A)* delay is invoked on a ground negative literal G_T when the tabling completion operation encounters a loop through negation involving G_T. Afterward, G_T is never re-evaluated, but instead simplified away when further evaluation of goals upon which G_T depends determines it to be *true* or *false*. On the other hand, instantiation delay occurs when a goal G_I is not instantiated enough to be evaluated; afterward, any time a variable in G_I is instantiated, G_I is rechecked to determine whether it satisfies an instantiation property (such as nonvar or ground) and evaluated if so.

In this section, we first show how ℰrgoAI automatically uses instantiation delay to address unsafe negation, and then how it can be invoked by the user to handle safety requirements imposed by external knowledge sources, or for other reasons, such as improving query performance.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

$\mathbf{P}_{unsafe}$

```
p(?X):- q(?X), \naf s(?Y), r(?X), t(?Z,?Y).
q(?X) :- \naf q2(?X).
r(?X) :- r2(?X).
q2(1).      r2(2).        s(3).         t(4,?).
```

```
Evaluation of p(?X):
1   p(?X):- q(?X), \naf s(?Y), r(?X), t(?Z,?Y).
    p(?X):- \naf s(?Y), r(?X), t(?Z,?Y),     // delay q2/1
3            when(ground(?X),\naf q2(?X)).
4   p(?X):- r(?X), t(?Z,?Y),                 // also delay s/1
             when(ground(?X),\naf q2(?X)), when(ground(?Y),\naf s(?Y)).
6   p(?X):- r2(?X), t(?Z,?Y),                //  r2/1 binds ?X to 2
             when(ground(?X),\naf q2(?X)), when(ground(?Y),\naf s(?Y)).
8   p(2):- \naf q2(2), t(?Z,?Y), when(ground(?Y),\naf s(?Y)).
    p(2):- t(?Z,?Y), when(ground(?Y),\naf s(?Y)).
10  p(2):- not_exists(s(?Y)). // fallback to XSB's well-founded not_exists/1
    No                                       // fail, since s(3) is true
```

Fig. 7. Schematic evaluation of non-ground default negation.

7.1 Unsafe negation in ℰrgoAI

Consider the resolution of the goal `p(?X)` using the set of rules and facts in Figure 7. Since by default ℰrgoAI uses a left-to-right literal selection strategy, the literal selected in Step 1 of the resolution is `q(?X)`, which is then resolved using a program rule to produce the subgoal `\naf q2(?X)`. Since `\naf q2(?X)` is an unsafe literal, ℰrgoAI attempts to *delay* the literal's evaluation in the hope that `?X` will be grounded later on in the evaluation – for example, the grounding of `?X` might happen after resolving `r(?X)` later in the rule for `p(?X)`.

The postponement is done using a modified version of the `when/2` predicate found in XSB and other Prologs, which in this case delays the goal `\naf q2(?X)` until `?X` becomes ground. This is shown on line 3 of the evaluation. The predicate `when/2` is based on *attributed variables*, which underlie the evaluation of logical constraints in Prolog systems. In the next step, `\naf s(?Y)` is selected and postponed, as shown on lines 4–5. Next, `r/1` is resolved with the third rule and then with `r2(2)`, which finally binds `?X` to 2, grounding the negative subgoal `\naf q2(?X)` to `\naf q2(2)`. This is shown on lines 6–8. Since `\naf q2(2)` is *true*, we remove it, reaching line 9. Note that grounding of `\naf q2(?X)` was requested on line 3 but did not happen until line 8, after `r2(?X)` was resolved. The two terms involved in this interaction, `\naf q2(?X)` and `r2(?X)`, do not even occur in the same rule. Moreover, when `\naf q2(?X)` was delayed, it was not known whether `?X` would ever become ground. The point is that the algorithm for subgoal delay is automatic and involves inter-rule interaction and run-time analysis.

Resolving `t(?Z,?Y)` leaves a non-ground subgoal `\naf s(?Y)`, which cannot be grounded later because the variable `?Y` does not appear outside of that subgoal. How should one proceed? In our experience, in most such cases, the user wants to check whether *no grounding* of `?Y` would ever make `s(?Y)` *true*. This can be efficiently verified

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

using the XSB predicate `not_exists/1`, as shown on lines 10 and 11 in Figure 7. Roughly, XSB defines `not_exists(G)` as `(TG, fail; tnot(TG))`, where `TG` is a tabled version of `G`, and `tnot/1` is XSB's ground well-founded negation. In certain cases, the user might want to assign the truth value u to non-ground `\naf` literals, such as `\naf s(?Y)`. This option can be selected in ℰrgoAI via a runtime switch.

Our discussion above considerably simplifies ℰrgoAI's actions; for instance the `\naf` operator also handles variables in *quantified* subgoals, based on extensions of Lloyd and Topor (1984). Furthermore, it is generally accepted that `\neg G` implies `\naf G` Alferes *et al.* (1994), and ℰrgoAI also uses this inference when grounding `\naf` -negated subgoals. We note that many aspects of ℰrgoAI's approach to `\naf` could be adopted by Prologs that implement WFS, such as SWI and XSB itself. In fact, nu-Prolog Naish *et al.* (1989) implements subgoal reordering for Prolog negation, although it does not assign the truth value *undefined* to non-ground negative subgoals, nor does it handle explicit negation or quantifiers.

7.2 Delay quantifiers

The delay quantifiers `wish/1` and `must/1` cause dynamic reordering of subgoals that do not fulfill argument binding requirements. These quantifiers check conditions on variables in a subgoal and delay that subgoal if these conditions fail. For example, the subgoal

```
must(ground(?X) or nonvar(?Y))^?X[foo->?Y]
```

contains the delay quantifier `must/1`, and evaluation of the subgoal delays evaluation of `?X[foo->?Y]` until either `?X` is ground or `?Y` is partially instantiated. If the desired instantiation is unachievable, a runtime error results. The `wish/1` quantifier delays its subgoal in the same manner as `must/1`, but the subgoal is executed after no more delay is possible, and no error is raised.

8 KRR in a dynamic setting

Most real-world knowledge changes over time. A `Claim` object, like that in Figure 3, may be added or modified. New rules expressing beliefs, like those in Figure 5, may be designed by a knowledge engineer or learned. Knowledge must also adapt to changing circumstances. For instance, if polls show that a viral pandemic is not a high concern of voters, a number of rules may be canceled in the sense of defeasible logic (Section 6).

In ℰrgoAI, *every* rule and fact is treated as dynamic information, so that any rule or fact may be inserted, deleted, enabled, or disabled.[21] The changes made possible by such flexibility can give rise to different issues. A new claim may need to be added *transactionally* so that either all necessary elements must be added or none may be. Adding or canceling a rule can have complex consequences especially when argumentation theories are used or when a change is made to a program's classification hierarchy. As a result, assurance may be needed that certain conditions hold or do not hold after

[21] Unlike in Prolog, where every rule/fact is either dynamic (deletable) or static, based on predicate-level declarations.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

an update. $\mathcal{E}$rgoAI addresses such issues through Transaction Logic Bonner and Kifer (1994) extended with integrity constraints.[22]

Returning to our running example about claims and beliefs, it may be useful to distinguish between two classes of people who frequently make claims: politicians and journalists. In fact, we might like to ensure that no person is both a politician and a journalist within the time frame covered by our knowledge base. It is easy to write a rule to check for this:

```
occupationConstr(?Per):-
   ?Per:Person,?Per[occupation->journalist],?Per[occupation->politician].
```

How can this constraint be enforced? One might try defeasible reasoning, as in Section 6, and set the two occupations in opposition, but that could result in such persons having no occupation whatsoever (depending on the argumentation theory). In such cases, it is usually better to impose database-style *integrity constraints* via this command:

```
?- +constraint{occupationConstr(?)}.
```

This activates a database constraint that tells $\mathcal{E}$rgoAI to check the query

```
?- occupationConstr(?Person).
```

after each transaction. In logic programming, such a constraint is considered to be violated if this query returns any answers: such answers are treated as objects that *violate* the constraint. In our case by executing

```
?- insert{'Bunky Muntner'[occupation->politician]}.
```

the above query returns a `Person` object that has `journalist` and `politician` both listed as occupations thereby violating the constraint. Various options are available to handle such cases – from simple warnings to transactional rollbacks to callbacks to routines for handling the constraint violation.

Transactional rollbacks happen automatically, if updates are written using *Transaction Logic* Bonner and Kifer (1994). At the practical level, this means that the user must use *transactional* update operators, that is `t_insert`, `t_delete`, `t_enable`, `t_disable`, etc., instead of `insert`, `delete`, `enable`, `disable`, etc. For instance,

```
?- t_insert{'Bunky Muntner'[occupation->politician]}.
```

Operationally, one can think of such updates as *backtrackable*, which provides the rollback functionality. Transactional updates are committed when a goal returns to the interpreter level, or when `commit@\db` is executed. A transactional update may be part of a complex query that relies on extensive tabling. To address this, whenever a transactional update is executed or rolled back (i.e., backtracked over), any inferences that have been made are reactively adjusted via $\mathcal{E}$rgoAI's *SLGR(A)*, which uses incremental tabling (cf. Swift (2014)), ensuring *view consistency* of inferences with respect to updates.

[22] $\mathcal{E}$rgoAI also supports two other mechanisms for monitoring or managing change: latent queries and alerts.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

As a second example, consider a toy ℰrgoAI program for graph coloring:

$\mathbf{P}_{\mathbf{Color}}$:

```
colorGraph :- \naf uncoloredNode(?).        /* stop if all nodes are colored */
colorGraph :- colorNode, colorGraph.        /* color graph recursively */
colorNode :-                                /* color one node */
    \naf coloredNode(?N),
    color(?C)
    \naf (adjacent(?N,?N2),colored(?N2,?C)),
    t_insert(colored(?N, ?C)).
```

$\mathbf{P}$_*color* colors a graph by recursively picking a node N, picking a color for N, and then ensuring that N is not adjacent to a node of the same color. If the adjacency constraint cannot be satisfied, the program backtracks to find a new color, or to recolor a previous node. In the latter case, updates to `colored/2` will be rolled back as necessary for the search to continue. (Of course, graph coloring is most efficiently done by a constraint system or SAT-checker.) Within ℰrgoAI, backtracking through updates is more expensive than normal backtracking, which is performed efficiently by XSB's engine. However, the paradigm of using backtrackable updates together with incremental tabling can be useful for problems where the search space is highly complex and not well understood, such as analysis of malicious object code Schwartz *et al.* (2018).[23]

At the theoretical level, Transaction Logic is a higher-order logic with a model and proof theory. Specifically, it treats elementary updates, like insert and delete, as oracles, so that the approach can be applied to first-order logic with belief revision as well as to WFS with fact or rule updates. In either case, satisfiability is defined over *path structures* consisting of a sequence of program states separated by update operations. Thus semantically, finding a solution for a goal to $\mathbf{P}_{color}$ amounts to finding a path structure that is a model for the program and goal.[24] The result is that data updates can be used in an ℰrgoAI program like $\mathbf{P}_{color}$ without compromising ℰrgoAI's logical semantics.

ℰrgoAI's support for management of changing knowledge has a number of other features such as bulk transactional updates and rule updates. In fact, from the user's point of view, ℰrgoAI handles transactional updates of rules in nearly the same manner as facts, supporting transactionality, view consistency, integrity constraints and other features. However there is one difference. Because some ℰrgoAI rules are compiled into static Prolog code that cannot be updated, ℰrgoAI gets hold of the rules not by their syntax but rather through their rule ids (cf. Section 3). For this purpose, it provides two predicates, `t_enable`{*RuleId*} and `t_disable`{*RuleId*},

[23] This work was begun before ℰrgoAI's transaction logic was stable, so the paper uses XSB's *SLGR* resolution directly, with the application program itself logging and rolling back updates.

[24] In this and other respects, Transaction Logic, which models data updates and belief revisions, differs from modal logics such as dynamic logic Pratt (1978) that model accessibility of program states using a semantics of Kripke structures.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

9 Termination and bounded rationality in $\mathcal{E}$rgoAI

As discussed in Section 2 $\mathcal{E}$rgoAI's use of *SLGR* resolution ensures termination and produces the correct well-founded model for programs with the bounded term depth property. This termination class includes Datalog and is far larger than the termination class of Prolog using SLDNF. However, assuming queries are side-effect free, runtime $\mathcal{E}$rgoAI options can guarantee correct termination for any query to a program that has a finite model, and *sound* termination on any query to any program.

9.1 Termination on programs with finite models

Section 2 provided examples of SCCs of mutually dependent subgoals that correspond to unfounded sets in WFS. Another type of unfounded set is an *unfounded chain*, which can be illustrated by the simple $\mathcal{E}$rgoAI program:

$\mathbf{P_{FinModel}}$:

```
p(?X) :- p(f(?X))
```

Using $\mathcal{E}$rgoAI's default resolution method, *SLGR*, the goal `p(a)` to $P_{FinModel}$ produces an infinite chain of goals, `p(`f`(a))`, `p(`f`(`f`(a)))`, `p(`f`(`f`(`f`(a))))`, and so on. This is because *SLGR* creates a new table entry for any goal that is not a variant of a goal already in the table (cf. Section 2). $\mathcal{E}$rgoAI can address such situations via *subgoal abstraction*, in which subgoals are abstracted once they reach a maximum size. In $P_{FinModel}$, abstraction with subgoal depth 2 or more would rewrite any goal that unified with `p(`f`(f(?X)))` to `p(f(?X))`, ensuring that the evaluation will create only a finite number of subgoals.

The use of subgoal abstraction guarantees powerful semantic properties. Let us represent a three-valued well-founded model M in terms of two sets: the set of atoms that are *true* in M and the set of atoms that have truth value u in M. Since the well-founded model of $P_{FinModel}$ has no atoms with truth value *true* or u, its three-valued model is finite. In Riguzzi and Swift (2014), it is shown that if SLG is extended with subgoal abstraction then it correctly terminates for any program with a well-founded model that is finite under the above definition.

In $\mathcal{E}$rgoAI, subgoal abstraction can be requested with the help of the `goalsize` restraint in the `setruntime` directive. Loading into $\mathcal{E}$rgoAI the small fragment below,

```
?- setruntime{goalsize(4,abstract)}.
p(?X) :- p(f(?X)).
p(b).
p(f(c)).
?- p(?X). // the query
```

produces the following answers and terminates:

```
?X = b
?X = c
?X = f(c)
```

Without subgoal abstraction (*i.e.*, if the directive

```
setruntime{goalsize(4,abstract)}
```

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

is commented out) the above program will not terminate and will return no answers. In our case, however, the `goalsize` parameter tells the query subsystem to abstract subgoals larger than size 4, which forces termination.[25] Subgoal abstraction is implemented by XSB's tabling subsystem. Whenever a tabled goal is encountered, a 'subgoal check/insert' operation traverses the goal in order to determine whether an existing table may be used or a new table must be created. The declaration for subgoal abstraction activates an engine-level hook that abstracts the goal as part of the traversal Riguzzi and Swift (2014).

9.2 Sound termination on programs with infinite models

A different source of non-termination arises from programs with infinite three-valued models, as when the query `p(?X)` is made to the following program.

$\mathbf{P_{InfModel}}$:

```
p(f(?X)) :- p(?X).
p(a).
```

Sound termination for WFS is possible using *restraint* Grosof and Swift (2013) a type of answer abstraction that sets the truth value of abstracted answers to u. For instance, when using restraint at term depth greater than or equal to two, the query `p(?Y)` posed to $P_{InfModel}$ would produce `p(a)` and `p(`f`(a))` both as *true* answers, along with `p(`f`(f(?X)))` with truth value u. In this way, an informationally sound approximation to the goal `p(?Y)` is provided. Specifically, when using restraint to evaluate a query to a program P, all atoms proved *true* or *false* by ℰrgoAI are *true* or *false* in the well-founded model of P. However some atoms that are *true* or *false* in the well-founded model may be u in ℰrgoAI. Through restraint, ℰrgoAI supports a fully semantic approach to bounded rationality.

Answer abstraction can be requested with the help of the `answersize` restraint in the `setruntime` directive. Loading into ℰrgoAI the small program below,

```
?- setruntime{answersize(3,abstract)}.
p(f(?X)) :- p(?X).
p(a).
?- p(?X).  // query
```

produces the following answers:

```
?X = a
?X = f(a)
?X = f(f(a))
?X = f(f(f(a)))
?X = f(f(f(?_h1945(?_h1946))))  - undefined
```

Without the answer abstraction (i.e., without the `setruntime` directive), the above query does not terminate.

[25] The actual size is computed by taking into account the function symbols in the goal with their arities in a prefix-style traversal that preserves termination properties.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

9.3 Tripwires

From the viewpoint of $\mathcal{E}$rgoAI, both subgoal abstraction and restraint based on answer abstraction are instances of *tripwires*. A tripwire allows a developer to specify that a property of an evaluation should cause some intervention from the system. The intervention may be an automatic change in the resolution method, such as with call abstraction or restraint; or the intervention could throw an error; or suspend the computation for debugging. $\mathcal{E}$rgoAI also supports a *max answers* tripwire that allows a user to set a limit on the number of answers a given subgoal S may have. If this limit is reached, a new answer A_S is added that is a variant of S, the truth value of A_S is set to u and S is completed. Afterward, queries to S will return all of the derived answers for S, as well as A_S. The effect is to change the closed-world assumption for S from using *false* as default to using u as default. Another kind of tripwire is the ability to time out a goal G_{costly} that may either call $\mathcal{E}$rgoAI or an external system. The tripwire can be set either to add an answer A that is a variant of G_{costly} with truth value u, or to throw an exception. Finally, as mentioned in Section 7, a negative subgoal that cannot be ground via delaying its execution can be given the truth value u instead of throwing an error.

In each of these types of tripwire a developer can weigh the benefit of a bounded execution time against the possibility of deriving a set of answers that are informationally sound but not complete. It is because of such uses that we denote the third WFS truth value as u, since it can mean *unknown* in addition to *undefined*.[26]

10 Architectural summary

Figure 8 illustrates the high-level architecture related to the features presented in previous sections. $\mathcal{E}$rgoAI is compiled into tabled Prolog code that is executed by the XSB engine, along with $\mathcal{E}$rgoAI's runtime libraries. Optionally Janus (Section 3.2) can load a Python engine into the same process, leading to extremely efficient calls between the $\mathcal{E}$rgoAI and Python. Direct interfaces to Java, C, SQL databases, SPARQL, and direct connectors to XML, JSON, and RDF are also available.

More specifically, the $\mathcal{E}$rgoAI compiler performs the HiLog and frame transformations of Section 4 and compiles them (Section 3) first into GPP macros and then, at load time, further into Prolog. The first of these steps is performed by the $\mathcal{E}$rgoAI compiler and the second by the GPP preprocessor. Most of these macros provide a convenient way to specify how module-dependent *salting* is to be done.

Inheritance and other background theories, such as typing, are implemented using $\mathcal{E}$rgoAI builders of GPP-enhanced Prolog as an intermediate language. In addition, the compiler performs transformations in support of several aspects of defeasible reasoning (Section 6.1). This includes handling rule tags and other rule descriptors as well as adding a defeasibility check to each defeasible rule. The resulting Prolog code makes use of *SLGR(A)* tabling (Section 2), attributed variables for instantiation and `\naf` delay (Section 7) and tripwires for bounded rationality (Section 9).

[26] In the case of procedural and other non-logical modules, the action associated with a tripwire can be to throw an error, or to call a user-defined handler.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

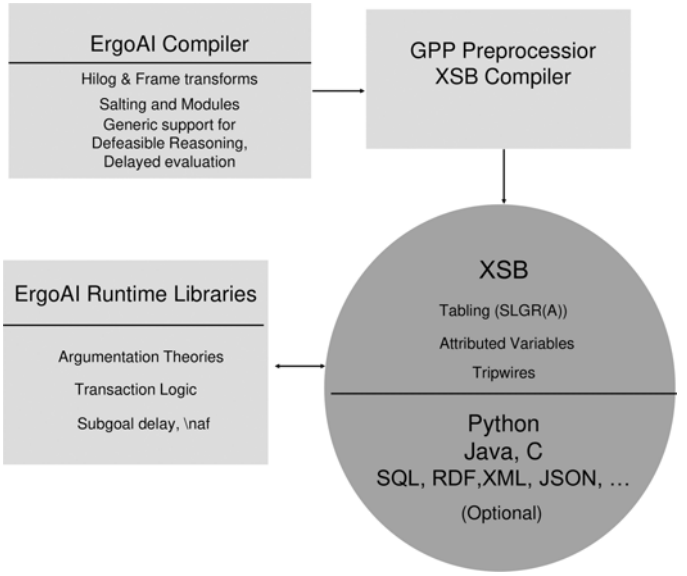


Fig. 8. A schematic architecture for ℰrgoAI.

11 Scalability and performance

Because ℰrgoAI is based on WFS it can be expected to be more scalable than KRR systems such as ASP or description logic solvers on transitive closure and other WFS-style queries. Indeed, a recent paper benchmarks transitive-closure queries, comparing XSB's tabling both to the ASP solver Clingo and to the Datalog system Soufflé, showing excellent performance by tabling over a variety of graphs Liu et al. (2025). One reason for this is that XSB, like ℰrgoAI, does not require grounding, which is inefficient for large graphs. On the other hand, while ASP problems such as Hamiltonian path and graph coloring can be directly coded in ℰrgoAI in a declarative manner, ℰrgoAI's search strategy will perform worse than that of Clingo or other ASP systems.

Building on these results, we compare ℰrgoAI's scalability and performance to that of XSB's tabling. We demonstrate that despite the overhead of HiLog, frames, instantiation delay, support for dynamic KRR, and other features, ℰrgoAI remains within an order of magnitude of tabled Prolog.[27] We begin with a benchmark of Petri Net reachability that indicates the general scalability of ℰrgoAI. We then separately benchmark tabled and non-tabled programs. A contains code for all benchmarks.

11.1 Performance comparison for petri net reachability

Our first benchmark addresses the problem of reachability in 1-safe Petri Nets, to which the central problems of liveness, deadlock-freedom, and the existence of home states can

[27] All benchmarks in this section were performed on a 2023 MacBook Pro with an Apple M2 Pro chip and 16 Gbyte RAM running Sonoma 14.7.2.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

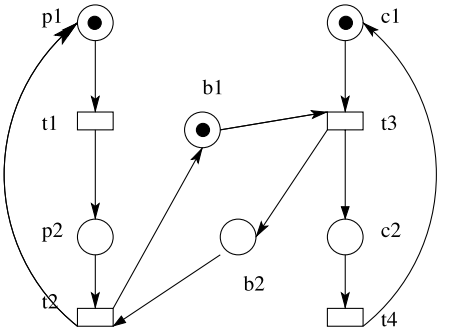


Fig. 9. Illustration of a 1-safe petri net.

States	400,000	800,000	4,000,000	8,000,000	40,000,000
$\mathcal{E}$rgoAI Time	9.4 sec	18.7 sec	97.2 sec	202.0 sec	*
XSB Time	1.7 sec	3.1 sec	19.5 sec	42.9 sec	244.0
$\mathcal{E}$rgoAI Memory	0.64 GB	1.26 GB	6.22 GB	12.4 GB	*
XSB Memory	0.16 GB	0.31 GB	1.46 GB	2.9 GB	14.8

Fig. 10. $\mathcal{E}$rgoAI /XSB elapsed time and space for petri net reachability.

be reduced (cf. Rozenberg and Engelfriet (1998)). The benchmark, from Marques *et al.* (2008), combines tabled and non-tabled code,[28]

A 1-safe Petri Net is a digraph of places and transitions: in Figure 9 the set of places is {*p1,p2,b1,b2,c1,c2*} while the set of transitions is {*t1,t2,t3,t4*}. Transitions are represented as facts such as:

```
transition([b1,c1],[b2,c2],t1)
```

indicating that if tokens are in places `b1` and `c1`, then transition `t1` can fire, placing tokens in `b2` and `c2`. The state of a given Petri Net is represented as a list of the places that contain a token: for example, in Figure 9 the state is represented as `[b1,c1,p1]`.

Figure 10 compares $\mathcal{E}$rgoAI and XSB for reachability queries to various sizes of generated Petri Nets. Note that both the time and space for $\mathcal{E}$rgoAI scale nearly linearly to a net with 8 million states, after which point RAM contention becomes an issue. XSB's elapsed times are 4.7–6.0 times faster than $\mathcal{E}$rgoAI's, while XSB's space is 4.0–4.7 times smaller than $\mathcal{E}$rgoAI's.

11.2 Performance comparison for tabled resolution

Left-recursive transitive closure is a good benchmark for tabling, since tabling performs nearly all computation apart from accessing the graph edges. Figure 11 shows tests over a chain, a cycle, and a chain in which all vertices have a self-loop. For $\mathcal{E}$rgoAI, measurements are performed for left-recursion over both HiLog predicates and frames. In these evaluations, left-recursion in frames incurs a 20–40% time overhead over predicates in $\mathcal{E}$rgoAI.

[28] The XSB distribution includes a suite of Petri Net and Workflow Net benchmarks some of which rely on constraint handling or well-founded negation.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

Graph	Vertices	HiLog Time	Frame Time	XSB Time	HiLog Space	Frame Space	XSB Space
chain	10^6	0.21/0.23	0.26/0.27	0.08/0.09	0.14 GB	0.16 GB	0.04 GB
chain	10^7	1.95/2.08	2.66/3.04	0.57/0.64	1.4 GB	1.45 GB	0.47 GB
chain	10^8	20.1/34.9	27.5/60.2	5.43/5.90	14 GB	14.2 GB	4.5 GB
self-loop	10^6	0.39/0.40	0.47/0.49	0.11/0.12	0.14 GB	0.16 GB	0.04 GB
self-loop	10^7	3.68/3.82	4.85/5.03	0.94/0.98	1.4 GB	1.45 GB	0.47 GB
self-loop	10^8	37.4/49.4	48.8/69.9	9.23/9.71	14 GB	14.2 GB	4.5 GB
cycle	10^6	0.21/0.23	0.25/0.26	0.08/0.09	0.14 GB	1.65 GB	0.04 GB
cycle	10^7	2.31/3.79	2.61/2.74	0.58/0.64	1.4 GB	1.45 GB	0.44 GB
cycle	10^8	23.6/40.0	26.8/47.3	5.59/6.06	14 GB	14.2 GB	4.5 GB

Fig. 11. CPU time/elapsed time in seconds, and space for left-recursive transitive closure.

```
makelist(0,[]):-!
makelist(?N,[?N|?Rest]):-
    ?N1 \is ?N-1,
    makelist(?N1,?Rest
```

```
iter(0):-!.
iter(N):-
     ?N1 \is ?N-1,
     iter(?N1).
```

Fig. 12. Test predicates to compare non-tabled resolution.

This is because a frame query `Id[Attr->?Val]` must check for answers through inheritance in addition to those obtained through transitive closure. For predicates, ℰrgoAI's speed is consistently 3–4 times slower than XSB. The reason is that the table space used by ℰrgoAI is itself 3–4 times that of XSB due to ℰrgoAI's explanation structures.[29] CPU time scales linearly up to graphs with 100–200 million edges on this platform. Elapsed time has more variance, especially when the memory needed for table space by ℰrgoAI becomes nearly the size of RAM (e.g., 14.5 Gbytes required out of 16 Gbytes available).

11.3 Performance comparison for non-tabled resolution

General programming in ℰrgoAI supports an open approach to knowledge formats, communicates with other components in a software stack, drives user interfaces, and enables much other functionality. ℰrgoAI is well equipped to support general programming because it does not need to ground the entire program to evaluate a query and because it supports data structures through logical terms. We use two simple predicates to benchmark general programming: `makelist/2` to construct a list, and `iter/2` to iterate through a loop (Figure 12). Both predicates perform structural recursion and do not need tabling.

The ℰrgoAI declaration *:- prolog{f/n}* causes a predicate to be compiled without tabling, although *f/n* is compiled with ℰrgoAI features such as the ability to be disabled (Section 8) and with two explanation arguments.

We call such rules *quasi-Prolog rules*. As noted previously ℰrgoAI can also call and be called from XSB Prolog, which provides another means to execute gemeral programming tasks.

[29] This paper does not discuss ℰrgoAI's explanation mechanism which is in the process of being redesigned and re-implemented. See Chapter 52 of the *ℰrgoAI Reasoner User Manual* for a full description.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

Execution Method for `makelist/1`	100,000	1,000,000	10,000,000
$\mathcal{E}$rgoAI and XSB (static)	< 0.01	0.04	0.46
$\mathcal{E}$rgoAI quasi-Prolog rules	0.12	1.63	31.9
XSB alone (static)	< 0.01	0.02	0.18
XSB alone (dynamic)	0.02	0.17	1.76
$\mathcal{E}$rgoAI quasi-Prolog rules (no heap gc)	0.07	0.74	10.0
XSB (dynamic, no heap gc)	< 0.01	0.08	0.82

Fig. 13. Elapsed times in seconds to create lists of varying lengths using `makelist/2`.

Execution method for `iter/1`	100,000	1,000,000	10,000,000
$\mathcal{E}$rgoAI and XSB (static)	< 0.01	0.02	0.14
$\mathcal{E}$rgoAI quasi-Prolog rules	0.06	0.33	3.64
XSB alone (static)	< 0.01	0.02	0.14

Fig. 14. Elapsed times in seconds for varying iterations using `iter/1` in $\mathcal{E}$rgoAI and/or XSB.

Figures 13 and 14 compare these $\mathcal{E}$rgoAI execution methods with both static and dynamic Prolog code, where static code undergoes a deeper compilation process than dynamic code, but unlike dynamic code cannot be modified. Figure 13 compares these execution methods on `makelist/2`. All methods are able to scale to lists with ten million elements. $\mathcal{E}$rgoAI calling static XSB code takes about the same time as XSB calling static XSB code. These methods are faster than both $\mathcal{E}$rgoAI quasi-Prolog rules and XSB dynamic code. However $\mathcal{E}$rgoAI quasi-Prolog rules do not scale linearly for `makelist/2`. Analysis showed that non-linearity is due to the creation on the heap of explanation structures that contain large lists. Indeed, when XSB's heap garbage collection is turned off, $\mathcal{E}$rgoAI quasi-Prolog rules scale in a nearly linear manner. As confirmation, Figure 14 shows linear scaling for quasi-Prolog rules on `iter/1` which does not create large structures.

11.4 Comparison of internal data stores

$\mathcal{E}$rgoAI can access large factual knowledge bases in several ways, including:

- As full HiLog facts, loaded directly into $\mathcal{E}$rgoAI's internal knowledge store or accessed from external databases; or
- As Prolog facts, loaded into XSB's internal knowledge store; or
- As Prolog terms, accessed via Python connectors through Janus

As a scalability experiment, a KG of 11,847,275 triples from the DARPA AIDA project (cf. Section 6) was loaded into the knowledge stores of $\mathcal{E}$rgoAI and XSB.

- *Loading the KG as $\mathcal{E}$rgoAI HiLog facts*. Using $\mathcal{E}$rgoAI's `fastload` the KG was loaded at a rate of about 59,000 facts per second of elapsed time. $\mathcal{E}$rgoAI's `fastload` uses trie indexing Ramakrishnan *et al.* (1999).

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

- *Loading the KG directly into Prolog.* Using XSB's dynamic loading, the KG was loaded with full indexing on each fact (7 indexes) at a rate of about 137,000 facts per second of elapsed time.

Of course, ℰrgoAI also has its own interfaces for SQL, SPARQL, XML, RDF, and other data sources. For knowledge in virtually any other format,ℰrgoAI can access Python libraries through the high-speed Janus interface (Section 3.2).

To summarize the overall performance and scalability results, ℰrgoAI is nearly as fast and scalable as tabled Prolog, while providing numerous sophisticated KRR features. For many applications these features are well worth the cost in terms of expressiveness, ease of maintenance, and updatability. Put another way, ℰrgoAI is a KRR system with scalability and performance similar to a programming system.

12 Discussion

ℰrgoAI is a multi-paradigm logic programming system that makes radically different choices from virtually all other logic programming systems. We have presented a few of these features, including frames and inheritance for object-oriented logic programming based on a solid semantics; HiLog predicates and terms integrated into a powerful module system; argumentation theories, the handling of unsafe queries, logical approach to updates and management of change, along with support for termination and bounded rationality. We have not covered in detail important features such as explanation; non-termination analysis; user-defined functions; quantified logical formulas in rule bodies and heads; typed variables and type constraints; and many other features of the ℰrgoAI language.

ℰrgoAI is a significant system in itself, but also a testbed for features that may be suitable for Prologs or other logic programming paradigms. For instance, the implementation of HiLog (Section 4) may extend well to Prologs like SWI, whose just-in-time indexing can adapt to the extra indexing burden required by the HiLog and frame transforms (Definition 2). Transaction Logic (Section 8) with non-recursive update rules can be implemented in Prolog via backtrackable dynamic code with hooks to call integrity constraints. Indeed the implementation of ℰrgoAI has shown that backtrackable updates can be implemented on top of regular `assert/1` and `retract/1` quite efficiently. However, a more complete implementation of Transaction Logic requires tabling of the *execution traces* of transactions. This remains difficult to implement efficiently, although Fodor and Kifer (2010) is a good step in this direction. The F-logic frames and classification hierarchies, whose compilation into Prolog was outlined in Section 3.4 mainly requires compiler and reader extensions rather than engine modifications, particularly in Prologs with strong tabling and indexing.[30]

As mentioned in Section 1, ℰrgoAI has proven useful in domains such as financial compliance, tax accounting, and legal reasoning. More recently, in the DARPA CODORD project, ℰrgoAI code is generated from natural language narratives and policy documents by five different automated systems based on LLMs, ILPs, semantic parsing and other techniques. Our near-term work includes continued improvements in ℰrgoAI's speed and

[30] Tabling is not strictly necessary for all aspects of frames, but is highly desirable both to deal with recursion and to efficiently implement inheritance.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

robustness, as well as improvements in explanation generation. Longer-term objectives include supporting fuzzy and probabilistic reasoning techniques for HiLog predicates. The approach under design uses techniques from the PITA Riguzzi and Swift (2011) system. PITA, which was originally implemented for XSB, is based on a source-level transformation followed by execution that uses tabling and BDDs. An initial extension to $\mathcal{E}$rgoAI's HiLog predicates requires reimplementing PITA's source-code transformation within the $\mathcal{E}$rgoAI compiler.

References

Alferes, J., Damásio, C. V. and Pereira, L. M. 1994. A top-down query evaluation for well-founded semantics with explicit negation. In *European Conference on Artificial Intelligence*. Morgan Kaufmann, San Francisco, CA, USA, 140–144.

Andersen, C., Benyo, B., Calejo, M., Dean, M., Fodor, P., Grosof, B., Kifer, M., Liang, S. and Swift, T. 2013. Understanding rulelog computations in silk. http://arxiv.org/abs/1308.4125

Blair, H. and Subrahmanian, V. S. 1989. Paraconsistent logic programming. *Theoretical Computer Science* 68, 135–154.

Bloomfield, R. 2024. Assuring transformative technologies: Assurance 2.0 and Clarissa tool chain. In *High Confidence Software and Systems Conference*. Science of Security Virtual Organization. URL: https://sos-vo.org

Bonner, A. and Kifer, M. 1994. An overview of transaction logic. *Theoretical Computer Science* 133, 205–265.

Brewka, G. and Eiter, T. 2000. Prioritizing default logic. In *Intellectics and Computational Logic – Papers in Honour of Wolfgang Bibel*. Kluwer Academic Publishers, Dordrecht, NL, 27–45.

Cer, D., Yang, Y., Kong, S., Hua, N., Limtiaco, N., John, R. S., Constant, N., Guajardo-Cespedes, M., Yuan, S., Tar, C., Strope, B. and Kurzweil, R. 2018. Universal sentence encoder for English. In *Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Stroudsburg, PA, USA, 169–174.

Chen, W., Kifer, M. and Warren, D. S. 1993. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming* 15, 3, 187–230.

Chen, W., Swift, T. and Warren, D. S. 1995. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming* 24, 3, 161–199.

Chen, W. and Warren, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM* 43, 1, 20–74.

CODORD 2024. Human-AI communication for deontic reasoning Devops. URL: https://www.darpa.mil/research/programs/codord.

Delgrande, J., Schaub, T. and Tompits, H. 2003. A framework for compiling preferences in logic programs. *Theory and Practice of Logic Programming* 3, 2, 129–187.

Dung, P. and Tran, S. 2001. An argument-based approach to reasoning with specificity. *Artificial Intelligence* 133, 1-2, 35–85.

Eiter, T., Faber, W., Leone, N. and Pfeifer, G. 2003. Computing preferred answer sets by meta-interpretation in answer set programming. *Theory and Practice of Logic Programming* 3, 4, 463–498.

Fodor, P. and Kifer, M. 2010. Tabling for transaction logic. In *Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 199–208.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

Gelfond, M. and Lifshitz, V. 1988. The stable model semantics for logic programming. In *Joint International Conference and Symposium on Logic Programming*. MIT Press, Cambridge, MA, USA, 1070–1080.

Gelfond, M. and Son, T. 1998. Reasoning with prioritized defaults. In *Interntional Conference on Logic Programming and Knowledge Representation*, Vol. 1471 in Lecture Notes in Computer Science. Springer, Berlin, DE, 164–223.

Grosof, B. 1997. Prioritized conflict handling for logic programs. In *International Logic Programming Symposium*. MIT Press, Cambridge, MA, USA, 197–211.

Grosof, B. and Swift, T. 2013. Radial restraint: A semantically clean approach to bounded rationality for logic programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press, Washington, DC, USA.

Kifer, M. 2022. Knowledge representation & reasoning with Flora-2. The Flora-2 Web Site. URL: http://flora.sourceforge.net.

Kifer, M., Lausen, G. and Wu, J. 1995. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM* 42, 741–843.

Kifer, M. and Lozinskii, E. 1992. A logic for reasoning with inconsistency. *Journal of Automated Reasoning* 9, 2, 179–215.

Liu, A., Stoller, S., Idogun, J. and Tong, Y. 2025. Efficiency of analysis of transitive relations using query-driven, ground-and-solve, and fact-driven inference. https://arxiv.org/pdf/2504.21291

Lloyd, J. and Topor, R. 1984. Making prolog more expressive. *Journal of Logic Programming* 1, 3, 225–240.

Lloyd, J. W. 1984. *Foundations of Logic Programming*. Springer, Berlin Germany.

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

Marques, R., Swift, T. and Cunha, J. 2008. Extending tabled logic programming with multi-threading: A systems perspective. URL: https://cliplab.org/Conferences/CICLOPS-2008/CICLOPS-2008-proceedings.pdf.

Morris, J. 2018. How programming can make the law more accessible. TEDxUAlberta.

Naish, L., Dart, P. and Zobel, J. 1989. The NU-Prolog debugging environment. In *International Conference on Logic Programming*. MIT Press, Cambridge, MA, USA, 521–536.

NIST 2022. Covid-19 global pandemic scenario for DARPA AIDA program.

Nute, D. 1994. Defeasible logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxfor dUniversity Press, London, UK, 353–395.

Phi3. 2024. Phi-3 technical report: A highly capable language model locally on your phone. arXiv:2404.14219.

Prakken, H. 1993. An argumentation framework in default logic. *Annals of Mathematics and Artificial Intelligence* 9, 1-2, 93–132.

Pratt, V. 1978. A practical decision method for propositinal dynamic logic. In *ACM Symposium on Theory of Computing*. ACM, New York, NY, USA, 326–337.

Przymusinski, T. 1990. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae* 13, 4, 445–465.

Ramakrishnan, I. V., Rao, P., Sagonas, K., Swift, T. and Warren, D. S. 1999. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming* 38, 1, 31–55.

Riguzzi, F. and Swift, T. 2011. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Theory and Practice of Logic Programming* 11, 4-5, 433–449.

Riguzzi, F. and Swift, T. 2014. Terminating evaluation of logic programs with finite three-valued models. *ACM Transactins on Computational Logic* 15, 4, 1–38.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

Rozenberg, G. and Engelfriet, J. 1998. Elemenary net systems. In *Lectures on Petri Nets I: Basic Models*, Vol. 1491, Springer, Berlin, DE, 12–121. LNCS.

Sagonas, K., Swift, T. and Warren, D. S. 2000. The limits of fixed-order computation. *Theoretical Computer Science* 254, 1-2, 465–499.

Sakama, C. and Inoue, K. 2000. Prioritized logic programming and its application to commonsense reasoning. *Artificial Intelligence* 123, 1-2, 185–222.

Schwartz, E., Cohen, C., Duggan, M., Gennari, J., Havrilla, J. and Hines, C. 2018. Using logic programming to recover C++ classes from compiled executables. In *ACM Conference on Computer and Communications Security*. ACM, New York, NY, USA.

Swift, T. 2014. Incremental tabling in support of knowledge representation and reasoning. *Theory and Practice of Logic Programming* 14, 4-5, 553–567.

Swift, T. and Andersen, C. 2023. The Janus system: Multi-paradigm programming in Prolog and Python. In *International Conference on Logic Programming*. EPTCS. https://www.eptcs.org

Swift, T. and Kifer, M. 2024. Multi-paradigm logic programming in the ErgoAI system. In *Interntional Conference on Logic Programming and Non-Monotonic Reasoning*, Vol. 15245 in Lecture Notes in Computer Science. Springer, Berlin, DE, 126–139.

Swift, T. and Warren, D. 2012. XSB: Extending the power of Prolog using tabling. *Theory and Practice of Logic Programming* 12, 1-2, 157–187.

van Gelder, A. 1989. The alternating fixpoint of logic programs with negation. In *ACM Principles of Database Systems*. ACM, New York, NY, USA, 1–10.

van Gelder, A., Ross, K. and Schlipf, J. 1991. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM* 38, 3, 620–650.

Wan, H., Grosof, B., Kifer, M., Fodor, P. and Liang, S. 2009. Logic programming with defaults and argumentation theories. In *International Conference on Logic Programming*. Springer, Berlin, DE, 432–448.

Wan, H., Kifer, M. and Grosof, B. 2014. Defeasibility in answer set programs with defaults and argumentation rules. *Semantic Web Journal* 6, 1, 81–98.

Wang, K., Zhou, L. and Lin, F. 2000. Alternating fixpoint theory for logic programs with priority. In *First International Conference on Computational Logic (CL'00)*, Vol. 1861 in Lecture Notes in Computer Science. Springer, Berlin, DE, 164–178.

Yang, G. and Kifer, M. 2003. Inheritance in rule-based frame systems: Semantics and inference. *Journal on Data Semantics* 2800, 69–97.

Yang, G., Kifer, M. and Zhao, C. 2003. Flora-2: A rule-based knowledge representation and inference infrastructure for the Semantic Web. In *International Conference on Ontologies, Databases and Appl. of Semantics (ODBASE-2003)*. Lecture Notes in Computer Science, Vol. 2888, Springer, Berlin, 671–688.

Zhang, Y., Wu, C. and Bai, Y. 2001. Implementing prioritized logic programming. *AI Communications* 14, 4, 183–196.

Appendix A Code used for benchmarks in Section 11

A.1 Code used to benchmark 1-safe petri nets

The tabled Prolog program for Petri Net reachability (Figure A1) was introduced in Marques *et al.* (2008) and is available in the XSB distribution under the path `mttests/benches/petri`. Both the Prolog and the $\mathcal{E}$rgoAI version (Figure A2) perform left-recursive transitive closure in which the base case backtracks through states to which there is a direct transition from the input state. Because a state of a Petri Net

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

is represented as a list of the places that contain a token, determining the new states requires a small amount of list manipulation. The ℰrgoAI version differs in small ways from the tabled Prolog version. In particular, predicates that are not to be tabled (called *transactional predicates*) have names that begin with `%` Because there is a small overhead to transactional predicates in ℰrgoAI, some predicate unfolding has been done in ℰrgoAI compared to the tabled Prolog version. Note, however that XSB list library predicates such as `member/2`, `flatten/3`, and `sort/2` are called directly, using module qualification. Also note that aggregation predicates like Prolog's `setof/3` have a different syntax in ℰrgoAI.

```
:- table reachable/2.
reachable(InState,NewState):-
   reachable(InState,State),
   hasTransition(State,NewState).
reachable(InState,NewState):-
   hasTransition(InState,NewState).

hasTransition(State,NewState):-
   get_trans_for_conf(State,AllTrans),
    member(Trans,AllTrans),
   apply_trans_to_conf(Trans,State,NewState).

get_trans_for_conf(State,Flattrans):-
   get_trans_for_conf_1(State,State,Trans),
   flatten(Trans,Flattrans).

get_trans_for_conf_1([],_State,[]).
get_trans_for_conf_1([H|T],State,[Trans1|RT]):-
   findall(trans([H|In],Out,Tran),trans([H|In],Out,Tran),Trans),
   check_concession(Trans,State,Trans1),
   get_trans_for_conf_1(T,State,RT).

check_concession([],_,[]).
check_concession([trans(In,Out,Name)|T],Input,[trans(In,Out,Name)|T1]):-
   ord_subset(In,Input),
   ord_disjoint(Out,Input),!,
   check_concession(T,Input,T1).
check_concession([_Trans|T],Input,T1):-
   check_concession(T,Input,T1).

apply_trans_to_conf(trans(In,Out,_Name),State,NewState):-
   ord_subtract(State,In,Diff),
   flatten([Out|Diff],Temp),
   sort(Temp,NewState).
```

Fig. A1. Tabled prolog program for reachability in a 1-safe petri net.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

```
reachable(?InState,?NewState):-
   reachable(?InState,?State),
   hasTransition(?State,?NewState).
reachable(?InState,?NewState):- hasTransition(?InState,?NewState).

hasTransition(?State,?NewState):-
   %get_trans_for_state_1(?State,?State,?MidTrans),
   flatten(?MidTrans,?AllTrans)@\plg(basics),
   member(trans(?Input,?Out,?_Name),?AllTranss)@\plg(basics),
   ord_subtract(?State,?Input,?Diff)@\plg(ordsets),
   flatten([?Out|?Diff],?Mid1)@\plg(basics),
   sort(?Mid1,?NewState)@\plg.

%get_rules_for_state_1([],?_State,[]):- \true.
%get_rules_for_state_1([?H|?T],?State,[?Trans1|?TransTail]):-
   ?TransSet = setof{?Trans | trans([?H|?Places],?Output,?Tran)@\plg(gen_elem),
                              ?Trans = trans([?H|?Places],?Output,?Tran) },
   %check_concession(?TransSet,?State,?Trans1),
   %get_rules_for_state_1(?T,?State,?TransTail).

// Check that a token is in all places in *t, and no token is in t*
%check_concession([],?_,[]):- \true.
%check_concession([rule([?_Inp|?Ilist],?Outlist,?_Name)|?T],?Input,
                  [rule([?_Inp|?Ilist],?Outlist,?_Name)|?T1]):-
   ord_subset(?Ilist,?Input)@\plg(ordsets),
   ord_disjoint(?Outlist,?Input)@\plg(ordsets),!,
   %check_concession(?T,?Input,?T1).
%check_concession([?_Rule|?T],?Input,?T1):-
   %check_concession(?T,?Input,?T1).
```

Fig. A2. $\mathcal{E}$rgoAI program for reachability in a 1-safe petri net.

```
tc_cycle(?Limit,?From,?To):-
    edge_cycle(?Limit,?From,?To)@\plg(tc_cycle).
tc_cycle(?Limit,?From,?To):-
    tc(?Limit,?From,?Mid),
    edge_cycle(?Limit,?Mid,?To)@\plg(tc_cycle).
```

Fig. A3. $\mathcal{E}$rgoAI code for transitive closure over large cycles.

A.2 Code used for benchmarking tabled resolution

Figures A3 and A4 show the code used in benchmarks for transitive closure. Note that the $\mathcal{E}$rgoAI transitive closure code calls the Prolog `edge/2` predicate (Section 11.2). The code is shown for cycles only; code for chains and chains with self-loops is similar.

A.3 Code used for benchmarking non-tabled resolution

$\mathcal{E}$rgoAI quasi-prolog rules. In a module for which production mode is on, $\mathcal{E}$rgoAI quasi-prolog rules are declared via `?- prolog{/1}`, which causes compilation to omit the HiLog transformation and ensures that the rule is not tabled. For these experiments, subgoal delay was manually turned off for these rules[31] but extra arguments, used for explanation, are added to the compiled version of the predicates.

[31] Automatic subgoal delay is not needed for Prolog predicates and can be expensive for large terms, such as those created in `makelist/2`.

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press

```
:- export tc_cycle/3, edge_cycle/3.

:- table tc_cycle/3.
tc_cycle(Limit,From,To):- edge_cycle(Limit,From,To).
tc_cycle(Limit,From,To):-
    tc_cycle(Limit,From,Mid),
    edge_cycle(Limit,Mid,To).

edge_cycle(Limit,From,To):-
    (Limit >= From ->
         (To is From+1 ; To=From)
      ;  fail).
```

Fig A4. Prolog code for transitive closure over large cycles.

```
:- compiler_options{production=on}.

?- disablefeature{subgoal_delay}.
:- prolog{makelist/2}.
makelist(0,[]):- !.
makelist(?N,[?N|?T]):-
   is(?N1 , ?N - 1)@\plg,
   makelist(?N1,?T).
```

```
:- prolog{iter/1}.
iter(0):-!.
iter(?N):-
   is(?N1 , ?N - 1)@\plg,
   iter(?N1).
```

Fig. A5. ℰrgoAI quasi-prolog rules used for benchmarking.

ℰrgoAI rules that call XSB. Calling XSB from ℰrgoAI is shown below.

```
makelist(?N,?L):-
   plg_makelist(?N,?L)@\plg (makelist).
```

```
iter(?N):-
    plg_iter(?N)@\plg (iter).
```

https://doi.org/10.1017/S1471068426100465 Published online by Cambridge University Press