A Framework for Answering Queries Using Multiple Representation and Inference Techniques

Nicholas L. Cassimatis
Naval Research Laboratory
4555 Overlook Dr. SW, Code 5513
Washington, DC 20375, USA
cassimatis@itd.nrl.navy.mil

Abstract

The Polylog framework is designed to provide a language for efficiently automating complex queries of information represented in multiple formats. A Polylog program contains a set of modules called *specialists* that store and make inferences about data in a particular representation. The FocusLoop algorithm answers queries by combining the knowledge and computation of all the specialists. *Logic program duals* for Polylog programs are introduced to prove that FocusLoop is sound and complete. A logic program dual makes the same inferences as the Polylog program it corresponds to. By using one program to formally characterize behavior and another to implement it, the traditional tradeoffs between provably correct automated question answering, representational flexibility and efficient execution are greatly reduced. Specialists using representations such as neural networks, ontologies, logical clauses and constraint graphs have already been implemented. They demonstrate that complex queries over multiple data sources can be automated without sacrificing efficiency for soundness and completeness. Finally, it is shown that FocusLoop generalizes logical deduction using operations such as resolution, forward inference and subgoaling and that these are common themes in many computational frameworks. In Polylog, each operation is implemented using multiple algorithms, enabling the weaknesses and impasses of one inference or representation technique to be compensated for by the strengths and resources the others.

1 Introduction

The increasing availability of large stores of data in multiple formats requires new tools for formulating and answering queries. Because each format (e.g., flat text file, XML, relational database, geographic information system) is best queried using its own specialized set of algorithms, executing complex queries that use multiple data sources is at best cumbersome. Web service protocols such as SOAP provide some relief in the form of a uniform programmatic interface to data, but they still require a

Polylog has grown out of the author's work on Polyscheme [4]. Thank you to Ed Hurley for a conversation about Polyscheme in the context of logic programming.

specific programming effort for each new kind of complex query that combines several web services.

One solution would be to design a query language that automatically answers queries over a wide variety of data formats. A problem with such query languages is that in order to provide sound and complete answers, they confine users to a single logical language and theorem proving algorithm so that mature semantic and proof theories for logical languges can be used to soundness and completeness.

Reliance on a single representation and algorithm is in tension with a common belief among computer scientists that the best computational techniques for solving a problem vary from domain to domain. Several trends in artificial intelligence are examples of the benefits of using multiple representation and inference techniques. For example, researchers in robotics find benefits in combining logical inference [10] or cognitively-plausible architectures [3] with robot mobility platforms. Researches modelling human cognition [8, 1] use multiple representations to capture the resourcefulness and complexity of human thought. Information retrieval research [2] achieves significant efficiencies using several representations for formulating and answering queries.

A specific consequence of relying on a single computational framework is the difficulty of integrating intelligent reasoning techniques with many kinds of languages, database, platforms and computer applications, all potentially distributed over a network of computers. Work integrating relational databases with logical programming systems [7] addresses this issue, but there are no general guidelines for integrating multiple kinds of computational resources.

A tension therefore exists between automatic, provably correct query answering and the benefits of integrating multiple inference and representation techniques. The Polylog framework has been developed to achieve both. Polylog allows programmers to declare knowledge using multiple representations and provides modules, called specialists, to make efficient inferences and solve problems using this knowledge. Although specialists can be implemented using a wide variety of techniques, we introduce the notion of a Prolog logic program dual for a Polylog program. Because the logic program dual is a Prolog program it enables a formal characterization of its corresponding Polylog program. Because the logic program dual is only used only to formally characterize a Polylog program while Polylog specialists use the more efficient data structures and algorithms, generally implemented in a nonlogical language, the formal benefits of Polylog programs do not imply a loss of efficiency.

Several other research programs address aspects of this integration problem. As already discussed, query languages based on logical programming provide sound and complete answers to queries but strongly limit the representational and computational techniques that can be used to answer queries. "Unified" architectures such ACT-R [1] or SOAR [6] attempt to provide frameworks for implementing multiple inference techniques in one system, but limit themselves to one or two representational formats. Knowledge interchange formats such as KIF [5] provide a way for sharing knowledge, but not for sharing computation. Internet-based web services provide a uniform framework for accessing information and computational resources over a network, but as of yet there is no way of automatically combining these to answer complex queries that

require a combination of these resources.

2 The Polylog Framework

A Polylog program is a collection of modules, called specialists. Each specialist uses the representation and inference techniques, not necessarily based on logic, that are most suited to their specialty. Programmers express a specialist's knowledge using a syntax tailored to each specialist's representation. The specialists communicate among each other with a simple predicate logic language. Queries are posed to the program using this language and the FocusLoop algorithm described in this section answers these queries.

Definition 1. Polylog Program. *A* Polylog program *consists of:*

- *A set of ordered pairs* $\{\ldots(Si, Ki)\ldots\}$ *where each* S_i *is a specialist and each* K_i *is a text string representing the knowledge of* S_i*;*
- *A* Query Specialist*,* S_Q*, which is a logic specialist (described below);*
- *A set of constants,* C*; and*
- *A set of variables,* V*.*

Each specialist in a Polylog program must implement several functions over literals and truth values. The set of truth values, TV, is $\{true, false\}$. Literals are of the form $p(t_1, \ldots, t_m)$, where $p \in C$ and $ti \in (C \cup V)$. All constants are denoted with strings beginning with alphabetic characters and all of the variables are denoted with strings beginning with "?" . A literal $L[s]$ is variant of $L[t]$ if there is a substitution which makes $L[s]$ equivalent to $L[t]$. Each specialist must implement the following functions. Any programming language may be used.

- $ReportTV(literal, tv)$. This method is used to report to a specialist the truth value, tv, other specialists assign to a literal.
- $StanceOn(literal)$. Returns true if the specialist can infer that *literal* is true and false otherwise.
- $GroundLiterals(literal)$. Returns a set of true ground literals that are equivalent to *literal* under substitution.
- $Subgoals(literal)$: Returns a set of literals whose truth value would help determine the truth value of *literal*.
- $Assertions(literal)$: Returns a set of ground literals that become true once *literal* is added to the specialist's knowledge.

These functions will be further constrained below to enable proofs of soundness and completeness.

A *logic specialist* implements the functionality normally expected of a logic programming system. Specifically, $GroundLiterals()$ returns literals that achieve a subgoal by ground substitution on the specialist's literals. $StanceOn()$ returns true for a literal if it is implied by the specialist's knowledge and the true literals it has learned through $ReportTV()$. Newly solved subgoals are returned through $Assertions()$. Two examples illustrate how specialists are implemented using a wide variety of representation techniques. In both cases, the specialists use three different representations. A logical representation, which is common to all the specialists in a program, allows the specialists to share information and make requests of each other. The declarative representation allows the programmer to easily and clearly express the knowledge for the specialist and the computational representation is used by the specialists to efficiently make inferences.

Logical	**Declarative**	**Computational**

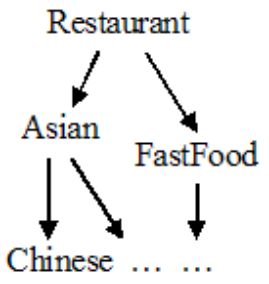
$Subcategory(Asian, Restaurant)$.
$Subcategory(FastFood, Restaurant)$.
$Subcategory(Chinese, Restaurant)$.
...
	`Restaurant`
	`contains:`
	`Asian,`
	`FastFood,`
	`...`
	
	`Asian`
	`contains:`
	`Chinese,`
	`...`
		

Figure 1. Representations used by the Ontology Specialist.

Logical	**Declarative**	**Computational**

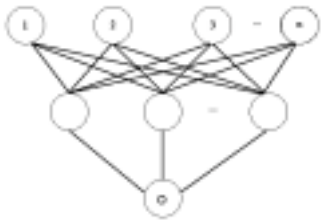
$Category(?r, Chinese)$
$Category(?r, FastFood)$
...
$NumberStars(?r, 0)$
$NumberStars(?r, 1)$...
$User(David)$
$User(Sarah)$...
	`INPUT UNITS:`
	`1: Category(?r,`
	`Chinese)`
	`2: Category(?r,`
	`FastFood) ...`
	`14: NumberStars(?r, 0)`
	`15: NumberStars(?r, 1)`
	`...`
	`26: User( David )`
	`27: User( Sarah ) ...`
	`OUTPUT UNITS:`
	`1: Likes(?user, ?r)`
	`WEIGHTS: [ .45 .67`
	`.26 .83 .93 ...`
	`.95 .23 .31 .07.49 ...]`
	`...`
		

Figure 2. Representations used by the Neural Network Specialist.

Figure 1 shows the representations for an ontology specialist. Its knowledge is declared by the programmer by enumerating of all the subcategories for each category, but it is represented internally using a directed graph. The ontology specialist

implements its functions in the obvious manner by using graph search algorithms. For example, when asked for a $StanceOn()$ a literal stating that $Chinese$ is a subcategory of $Restaurant$, it walks up the category graph from $Chinese$ until it either reaches $Restaurant$ or the root and returns the appropriate truth value.

Figure 2 shows the representations for a neural network specialist. The knowledge for this specialist is declared by asserting the propositions that are associated with each input and output unit and by providing numerical matrices that specify the connection weights for the network. The specialist implements its functionality by building instances of neural networks and firing neurons according to the weights provided by the programmer.

Input: A Polylog program, P, with specialists S_i, knowledge K_i, query specialist, S_q and its knowledge base, K_q and the alphabets of constants C and variables V; and query literals $Q_1, \ldots, Q_n$.
Output: A set, $Answers$, of the true ground literals whose arguments represent substitutions that make the Q_i true.

FocusLoop$(P, Q_1, \ldots, Q_n)$
empty $Foci$ and $Answers$ sets.
$QueryClause = (\sharp Answer(\ldots, v_i, \ldots) \leftarrow Q_1, \ldots, Q_n), n > 0.$
append $QueryClause$, to knowledge, K_q, of the query specialist, S_q.
$Query = \sharp Answer(\ldots, v_i, \ldots).$
add $QueryClause$ to $Foci$.
while $Foci$ is not empty.
  $FocusLiteral =$ arbitrarily selected element of $Foci$.
  remove $FocusLiteral$ from $Foci$.
  if $FocusLiteral$ is ungrounded
    for each specialist, S_i,
      add $GroundLiterals(FocusLiteral)$ to $Foci$.
  else
    $FocusTV = \vee_i(S.FocusTV(FocusLiteral))$.
    if $FocusTV$ is true and $FocusLiteral$ is a variant of $Query$
      add $FocusLiteral$ to $Answers$.
      for each specialist, Si,
        $ReportTV(FocusLiteral, FocusTV)$.
      add $\cup_i(S_i.assertion(FocusLiteral))$ to $Foci$.
    add $\cup_i(S_i.subgoals(FocusLiteral)$ to $Foci$.
remove $Query$ from the query specialist's knowledge, K_q.
return $Answers$

Figure 3. FocusLoop Algorithm

Polylog programs answer queries using the FocusLoop algorithm described in Figure 3. In the next section, it is proved that, when the specialists conform to certain

"duality conditions", FocusLoop's inferences are sound and complete. FocusLoop takes a Polylog program and a set of query literals and returns all the variable substitutions that would make the query literals true. FocusLoop proceeds by "focusing" on a series of literals until there are no new literals on which to focus. When a ground literal is focused on, all the specialists judge whether the literal is true or false. The combined truth values are reported to all the specialists and if a specialist needs to know the truth of another literal to offer a truth value on the current literal, it asks to focus on it. If the focus literal is ungrounded, all the specialists ask to focus on literals in their knowledge bases that ground that literal. When a specialist has enough information to infer that a literal is true, it asks FocusLoop to focus on it so that all the other specialists can make inferences based on that literal. This proceeds until all the literals requested by the specialist have been focused on. FocusLoop then returns all the substitutions from ground literals that answer the original query.

FocusLoop is superficially similar to resolution theorem proving algorithms because it continues to ask for subgoals until it reaches ground literals that are known to be true. When these are focuses on, it infers that the literals that follow are true. The crucial difference is that each step of resolution algorithms is performed with one kind of representation (clauses) and operated on by one kind of algorithm (resolution) and inference is essentially through one mechanism (material implication). In Polylog, each of these steps can be performed by multiple representations and algorithms. Section 4 will expand this contrast.

Some additional terminology will be useful. FocusLoop *focuses* on a literal each time it assigns *FocusLiteral* to it. A specialist *assents* to a literal if its *ReportTV* function returns *true* for it at least once during FocusLoop. FocusLoop *assents* to a literal if one of the specialists does. A specialist *grounds* Q to $Q[s]$ if it includes $Q[s]$ in the return set of its *GroundLiterals* function during a cycle when FocusLoop is focusing on Q. A specialist *asserts* a literal when it assents to the literal after having returned it in a previous call to its assertions function. FocusLoop asserts a literal when a specialists asserts it. A specialist *makes a subgoal* of a literal if a call to its subgoals function returns that literal at least once during FocusLoop. FocusLoop makes a subgoal of a literal when it focuses on it after a specialist makes it a subgoal. FocusLoop's *answer* for $QueryClause = (\sharp Answer(\ldots, v_i, \ldots) \leftarrow Q_1, \ldots, Q_n)$ is the set of all the literals in *Answers* after the FocusLoop terminates.

3 FocusLoop is Sound and Complete

In order to characterize the conditions under which FocusLoop is sound and complete, the notion of a *logic program dual (LPD)* for a Polylog system is introduced. An LPD for a Polylog system is a Prolog program that returns the same answers to queries that the Polylog system does. Because the semantic properties of Prolog are well understood, a Polylog system's LPD can be used to characterize its behavior, for instance, by establishing soundness and completeness while generally nonlogical programming languages can be used to most efficiently implement the Polylog system.

This paper deals with a subset of Prolog, Datalog, without identity, functions or

negation. In our notation, a Datalog clause is expressed, $P \leftarrow Q_1, \ldots, Q_n$, where P and all the Q_i are literals over the constant alphabet C plus $\sharp Answer$ and the variable alphabet V. P is the head of the clause and the Q_i form its body.

Because Prolog programs are sound and complete with respect to familiar first-order semantics, (e.g., as defined in [9]), this paper will prove that FocusLoop makes sound and complete inferences for a Polylog program by showing that it makes the same inferences as its LPD. Since the LPD is a Prolog program, FocusLoop's inferences will be sound and complete with respect to semantic model of the LPD. When a Prolog program, P, would answer that a literal, L, is true, we write $P \rightarrow L$. The Prolog program obtained by combining the literals and clauses of Prolog programs A and B is refered to as $(A \cup B)$.

Definition 2. Specialist Logic Program Dual. *An Prolog program, P_s, is a* logic program dual *(*LPD *or, simply,* dual*) for a specialist, S, if S does not assert or make a subgoal of any particular query or literal more than a finite number of times and if the following conditions, called the* duality conditions*, obtain at the end of each cycle of* FocusLoop*:*

- *If $(D_s \cup A) \rightarrow L$, where A is the set of all asserted literals so far, S will have asserted L during or before this cycle.*
- *If $H \leftarrow B_1, \ldots, B_n$ is a clause in P_s and* FocusLoop *has focused on $H[s]$, then S will have made, during or before this cycle, a subgoal of each literal in the set, $\{B_i[t]\}$, where the domain of t is a (nonstrict) subset of the domain of s and consistent with s otherwise.*
- *If Q is focused on and the ground literal $Q[s]$ is in P_s, then S will have grounded Q to $Q[s]$ at least once during or before this cycle.*
- *A specialist asserts or assents to literal, L, only if $(D_i \cup A) \rightarrow L$, where A is the program formed by the set of all asserted literals so far.*

Logic program duals for the example specialists discussed in the last section illustrate these ideas. The dual for a logic specialist is simply the program formed by the literals and clauses the programmer specifies for that specialist. The dual for the neural network specialist is a lookup table for the function the network produces. This table is formed by a series of clauses such as: $Likes(Robert, ?r) \leftarrow Category(?r, Chinese) + NumberStars(?r, 3) + User(Robert)$. This dual's computational inefficiency - it is impossibly large - is irrelevant because the computation is performed by the neural network, not its dual. This point is elaborated in section four. Finally, the dual for the ontology specialist is the list of *Subcategory* literals representing the subcategory relationships the programmer supplied for the specialist together with a transitivity clause: $Category(?o, ?c2) \leftarrow Category(?o, ?c1)$, $Subcategory(?c1, ?c2)$.

Definition 3. Program Logic Program Dual for Polylog Program. *A logic program dual for a Polylog Program is the union of the logic program duals for its specialists.*

A logic program dual is said to be *finite* iff it is comprised of a finite number of clauses and literals.

Theorem 1. Termination. *If a Polylog program has a finite dual, it terminates.*

Proof. FocusLoop continues only so long as $Foci$ is not empty. Each specialist can add a literal to $Foci$ a finite number of times. A specialist can only add a literal to $Foci$ if it is a literal (or one of a finite number of variants thereof) in the derivation tree for the query. Derivations trees in finite Prolog programs are finite, the number of specialists is finite and thus only a finite number of literals might ever be added to $Foci$. Because, at each cycle of FocusLoop at least one literal is removed from $Foci$, $Foci$ will ultimately be emptied and hence FocusLoop will terminate. □

Theorem 2. Soundness. *For a Polylog program, P, with a dual, D_P,* FocusLoop *assents to a literal, Q, only if $D_P \rightarrow Q$.*

Proof. FocusLoop assents to Q only if one of its specialists, S, assents to it. By the definition of a logic program dual, S assents to Q only if it is implied by S's dual, D_S. Because the literals and clauses of D_S are a subset of D_P and because D_P is consistent, then $D_P \rightarrow Q$. □

Answer completeness is proved first and then completeness will follow as a special case.

Theorem 3. Answer Completeness. *For a Polylog Program, P, with a consistent and finite dual, D_P,* FocusLoop *answers a query, Q, with the set of grounded literals, $\{\ldots Q[s_i] \ldots\}$ such that $D_P \rightarrow Q[s_i]$.*

Proof. Consider the Prolog derivation tree formed by ground substitution and resolution for each true ground literal $Q[s]$ that D_P implies. It is well known (e.g., [9]) that such a tree exists for every ground literal implied by a Prolog. The trunk of the tree is a ground literal $Q[s]$ and the leaves are ground literals. We prove that the leaves of every derivation tree for every answer for $Q[s]$ are asserted by FocusLoop and then show that once FocusLoop asserts the leaves of a derivation tree for $Q[s]$, FocusLoop will ultimately assert $Q[s]$.

Lemma 1. For each $Q[s]$ implied by M_P and derivation tree, T, for $Q[s]$, if FocusLoop focuses on variant $Q[t]$ of $Q[s]$, FocusLoop will assert the leaves of T.

Proof of Lemma 1. The lemma is proved by induction on T. Each interior node in T, by definition, is formed by a clause, C, of the form, $H[s] \leftarrow B_1[s] \ldots B_n[s]$ where $H[s]$ is either the trunk of a tree or a literal in the body of another clause in the tree. *Induction step.* If FocusLoop makes a subgoal of the head, $H[s]$, then FocusLoop will make a subgoal of variants, $B_i[t]$, of the body literals. This is because if D_P is the union of the logic program duals for the specialists, there must be a specialist, S, whose dual program, D_s, includes C. If FocusLoop makes a subgoal of $H[s]$, the specialist dual conditions require that the specialist makes a subgoal of variants of C's body, $B_i[t]$. *Base case.* The root of the T is $Q[s]$ and FocusLoop focuses on a variant,

$Q[t]$, by assumption. Each literal of the tree, therefore, will have a variant of it focused on by FocusLoop. In particular, each leaf of T, $L[s]$, has at least one variant, $L[v]$ that FocusLoop will make a subgoal of and focus on. The duality conditions for the specialists require that any specialist that contains $L[s]$ in its dual must ground $L[v]$ to $L[s]$ when $L[v]$ is focused on. Thus, the leaves of T will be asserted by FocusLoop.

Lemma 2. For each $Q[s]$ implied by D_P and derivation tree, T, for $Q[s]$, if FocusLoop asserts all the leaves of T, it will assert $Q[s]$ before it terminates.

Proof of Lemma 2. The lemma is proved by induction on the length of the path from the leaves to the root in T. *Induction step.* Let n be the number of interior nodes (or "steps") in T separating a node, $H[s]$, from $Q[s]$. $H[s]$ is formed by the clause, C, $H \leftarrow B_1, \ldots, B_n$. Each of the B_i is either a head of a clause in the tree or a branch of the tree and in either case is $n+1$ steps away from $Q[s]$. If all the literals $n+1$ steps away from $Q[s]$ have been asserted, then the specialist dual conditions on the specialist containing C imply that the specialist will assert $H[s]$. Hence, after all the literals n + 1 steps away from the answer, $Q[s]$, have been asserted, those literals n steps away will be asserted. *Base case.* Let N be the maximum number of interior nodes separating a leaf in T to $Q[s]$. Each literal N steps from the $Q[s]$ must be leaf (or its branches would be separated from $Q[s]$ by more than the maximum N nodes) and, by assumption, all the leaves are asserted. Thus, all literals which are separated by fewer than N nodes from the root of the tree will be asserted by FocusLoop. Because it is separated from itself by zero nodes, this therefore includes the trunk, Q[s].

These two lemmas together imply that FocusLoop is answer complete. For any query Q, with answers $Q[si]$ implied by D_P, and derivation tree T_i, lemma 1 implies that the leaves of T_i will be asserted and lemma 2 implies that once these are asserted, $Q[s_i]$ will be asserted. Because they are each true variants of Q, they will be part of the answer set. □

Theorem 4. *Completeness For a Polylog program, P, with a consistent and finite dual, D_P. If $D_P \rightarrow Q$, then* FocusLoop *will answer query Q with the set $\{Q\}$.*

Proof. If the query, Q, is a ground literal and D_P implies it, then, by answer completeness, it will be in the answer set. As the only ground variant to a ground literal is the ground literal itself, only Q will be answered. □

4 Relationship to Logical Deduction

The duality conditions on specialists, and hence FocusLoop, cleave so closely to the derivation tree for a logical program - and one formed through the comparatively inefficient process of ground substitution and resolution - that the specialized data structures and algorithms in a Polylog program do not seem to provide any gain in efficiency over Polylog program's logic program dual. Further, it appears as if programmers must construct correctness proofs for their programs, which is often

difficult for most commonly used programming languages. The explanation of why this is not so helps pinpoint the power of the Polylog framework and illustrates that Polylog is a form of generalized logical programming.

4.1 Large Duals, Short Trees and Fast Inference

Because a logic program dual is never executed, specialists can have duals with an intractably large proof space. For example, a Prolog program that would compute a *Reverse* predicate on lists would lead to derivations at least as long as the list and only then if the programmer is careful to consider the procedural implications of the structure and order of the program's clauses. However, a specialist that answers queries using a *Reverse* literal can have a dual that is simply a lookup table, i.e., a series of ground literals such as:

$\ldots$
$Reverse(zxcvbnm, mnbvcxz)$,
$Reverse(zxcvbnn, nnbvcxz)$,
$Reverse(zxcvbno, onbvcxz)$,
$\ldots$.

This logic program dual is too large to physically realize. The lookup table for *Reverse* will be exponentially larger than the longest list a computer's memory can hold and thus larger than computer's memory. Yet the model is easy to understand and the function it represents can be computed with a small amount of code.

Thus, by separating formal characterization from machine implementation and using logic programs to describe a specialist's functionality while using specialized representations and algorithms to implement that functionality, the Polylog framework achieves the ease, elegance and confidence of declarative programming with the efficiency and power of multirepresentational programming. Because logic program duals are often as simple as lookup tables, programmers of Polylog specialists must not generally be expert in the complexities of logic programming.

4.2 Generalizing Logic Programming

The one case where the logic program dual for a specialist is similar to the specialist's implementation is for specialists based on logic. Examining a logic specialist's functions will illustrate that Polylog generalizes the basic functions of a Polylog program and facilitates powerful and efficient programs.

Ground resolution. Logic programming systems often perform ground resolution by searching a list of ground literals. For cases where more efficient algorithms exist, Polylog allows specialists to implement these in the $GroundLiterals()$ function. For example, an arithmetic specialist might use a C function to resolve $Product(?x, 35, 105)$ or a geographical specialist can use spherical geometry to compute $Distance(Boston, Fargo, ?d)$.

Forward inference. Logic programs use material implication on a clause to infer

that its head is true if the literals in its body are true. A specialist's *Assertions*() function allows other algorithms to be used for forward inference when they are more efficient. For example, a neural network specialist computes the truth value of literals by applying a threshold function to an output unit's activation formed by network propagation.

Subgoals. When a logic program cannot retrieve the answer to a query it resolves a clause's head with the query and tries to prove the resulting literals in the clause's body. A specialist's *Subgoals()* function can use other algorithms when they are more appropriate. For example, when trying to determine if an object is of category C, the ontology specialist can use graph search algorithms to find all the categories that are ancestors of C in the category graph and make a subgoal of determining whether the object belongs to one of those categories.

Operations such as ground resolution, forward inference and subgoaling are thus not specific to logic programming. They are common themes that occur in many computational frameworks. FocusLoop reformulates logical derivations in terms of operations and allows specialists with other representation and inference techniques to implement these functions. The combined Polylog program is thus more powerful because weaknesses and impasses in one inference or representation technique are compensated for by the strengths and resources of another.

5 Conclusions

The Polylog framework is designed to attain the benefits of logical, declarative and multirepresentational programming in querying multiple sources of information in multiple formats. Given a set of specialized modules, program can be written by declaring knowledge using a wide variety of representations without concern for procedural details and rely on FocusLoop to answer queries. Logic program duals that mirror the functionality of Polylog specialists enable proofs that FocusLoop's answers are sound and complete so long as the specialists conform to certain duality conditions. Duality conditions also provide a set of guidelines for adding specialists to a program, preserving soundness and completeness. Polylog programs are fast and efficient because the functionality described by the logic program duals can be implemented using the (potentially nonlogical) techniques best suited for a domain. Specialists that have been implemented thus far demonstrate that Polylog enables the integration of intelligent reasoning techniques with the vast amount of information and computational resources available in other languages, on other platforms and on multiple machines over a network. FocusLoop reformulates logical deduction in terms of operations such as ground resolution, forward inference and subgoaling, shows that these are common themes in many computational frameworks and allows each of these functions to be fulfilled by many nonlogical representations and algorithms. Thus, Polylog provides efficient and robust programs by enabling the weaknesses of one inference or representation technique to be compensated for by the strengths of others.

References

[1] John R. Anderson and Christian Lebiere. *The Atomic Components of Thought*. Lawrence Erlbaum Associates, 1998.

[2] Nicholas J. Belkin, Colleen Cool, W. Bruce Croft, and James P. Callan. Effect of multiple query representations on information retrieval system performance. In *Research and Development in Information Retrieval*, pages 339–346, 1993.

[3] Magdalena D. Bugajska, Trafton Alan C. Schultz, Matthew Taylor J. Gregory, and Farilee E. Mintz. A hybrid cognitive-reactive multi-agent controller. In *In Proceedings of 2002 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 339–346, 1993.

[4] Nicholas L. Cassimatis. *Polyscheme: A Cognitive Architecture for Integrating Multiple Representation and Inference Schemes*. Ph.D. dissertation, Massachussetts Institue of Technology, 2002.

[5] Hans Chalupsky, Tim Finin, Rich Fritzson, Don McKay, Stu Shapiro, and Gio Weiderhold. An overview of kqml: A knowledge query and manipulation language. Technical report, KQML Advisory Group, April 1992.

[6] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[7] Jack Minker. Logic-based approach to data integration. *Theory and Practice of Logic Programming*, 2:293–321, May 2002.

[8] Marvin Minsky. *The Society of Mind*. Simon and Schuster, New York, New York, 1986.

[9] Ulf Nilsson and Jan Maluszynsi. *Logic, Programming and Prolog*. John Wiley and Sons, New York, New York, 2nd edition, 1995.

[10] Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, Cambridge, MA, 1995.