Converting Semantic Meta-knowledge into Inductive Bias*

John Cabral, Robert C. Kahlert, Cynthia Matuszek, Michael Witbrock, and Brett Summers

Cycorp, Inc., 3721 Executive Center Drive, Suite 100,
Austin, TX 78739
{jcabral, rck, cynthia, witbrock, bsummers}@cyc.com

Abstract. The Cyc KB has a rich pre-existing ontology for representing common sense knowledge. To clarify and enforce its terms' semantics and to improve inferential efficiency, the Cyc ontology contains substantial meta-level knowledge that provides definitional information about its terms, such as a type hierarchy. This paper introduces a method for converting that meta-knowledge into biases for ILP systems. The process has three stages. First, a "focal position" for the target predicate is selected, based on the induction goal. Second, the system determines type compatibility or conflicts among predicate argument positions, and creates a compact, efficient representation that allows for syntactic processing. Finally, mode declarations are generated, taking advantage of information generated during the first and second phases.

1 Introduction

Because of the general complexity of machine learning, the discipline has devoted significant attention to the use of inductive bias in improving algorithmic efficiency. Inductive logic programming (ILP) is in an interesting position on this topic because its declarative representations can be extended to include the representation of the ILP system itself, including the biases its algorithms employ. Tausend [1], [2], Nedellec et al. [3] have investigated the utility of declaratively representing biases as part of the background knowledge for ILP systems. They have noted that such representations allow for an explicit and modular representation of bias, such that biases used by different systems can be meaningfully compared and the settings of an individual system can be easily manipulated. McCreath [4], McCreath and Sharma [5], and DiMauro, et. al. [6] have begun the next phase of this work by developing algorithms for inducing type and mode biases from data. The implementation of these algorithms in pre-processors promises to reduce the burden on the human users of ILP systems. An alternative source for inductive biases is the language of the data, when that language is part of a larger ontology. Using the knowledge contained in the Cyc knowledge base (KB) – a very large, non-domain-specific ontology of formalized knowledge – provides a novel approach to automating the generation of inductive biases.

* Distribution Statement A: Approved for public release; distribution is unlimited.

S. Kramer and B. Pfahringer (Eds.): ILP 2005, LNAI 3625, pp. 38 – 50, 2005.

2 An Overview of the Cyc Project

The Cyc project is an ambitious, decades-long effort to generate a store of common-sense knowledge that enables reasoning in a broad array of domains. Human knowledge engineers have so far constructed most of the knowledge base (KB). However, one of the underlying premises of the Cyc project is that certain kinds of machine learning require the existence of a pre-existing body of knowledge into which new knowledge can be integrated (Lenat and Guha [7]). So, the long-term goal of the project is to invest human effort in the creation of a necessary foundation for effective, future machine learning. The goal of the research described in this paper is to begin the work of converting the existing knowledge into a basis for more effective automated learning. Since the representational language of Cyc is an extension of first-order predicate logic, inductive logic programming is the natural fit for the type of machine learning that would eventually drive the growth of the knowledge base.

The Cyc KB is represented using the language CycL, a LISP-like language that includes features like quantification over predicates and complete sentences, modal operators, and an extensive meta-language. All expressions are wrapped within parentheses and the leftmost entity within the parentheses being the predicate or function. Constants of the language are tagged with a prefixed '#$' and variables are composed of all capital letters and a prefixed '?'. All aspects of the language are represented within CycL. So, logical connectives follow the same syntactic rules. The following Prolog clause (a translation of "For any birth event E in which some M plays the role of female parent and some C is the child born during E, then M is C's biological mother"):

```
biologicalMother(C,M) :-
birthEvent(E), femaleParentActor(E,M), birthChild(E,C)
```

would be represented in CycL as:

```
(#$implies
(#$and
(#$isa ?BIRTH #$BirthEvent)
(#$femaleParentActor ?BIRTH ?MOTHER)
(#$birthChild ?BIRTH ?CHILD))
(#$biologicalMother ?CHILD ?MOTHER)).[1]
```

The above example illustrates a significant feature of the Cyc ontology. First, classes are denoted with constant names, not predicates. This choice is based on certain requirements of ontological engineering. First, it allows for multiple instance-to-class relations to be introduced into the vocabulary. Thus, we can specialize the predicate to specific types of collection to more precise relations such as occupations, ethnicities, and nationalities. Introducing these more specific relations increases the precision of the language and, as a consequence, allows for greater inferential

[1] When variables are unbound, the system assumes universal quantifiers with scope over the entire sentence.

efficiency because problems can be more narrowly defined. Second, because there is a significant inferential cost to reasoning with unbound predicates and it is quite common quantify over classes, introducing classes as named entities allows rules to be written at the right level of generality without incurring a significant computational cost.

The Cyc KB is structured using a hierarchical arrangement of microtheories. Microtheories represent contexts in which sentences are true. Microtheories inherit content from other microtheories to which they are linked. So, if MT_1 is more general than MT_2 than everything that is true in MT_1 is also true in MT_2. This relationship among microtheories is transitive and reflexive, but not symmetric. Sentences that hold universally are those which are expressed in the "highest," most general microtheories, while sentences that are true in very limited contexts (i.e., the content of a person's testimony or a work of fiction) are represented in microtheories that are much "lower" in the hierarchy. The great advantage of microtheories is that contradictory assertions can be represented within the KB without introducing formal contradictions because those assertions can be represented in microtheories that are not linked to one another.

As of March 2005, the Cyc KB contains over 296,000 reified terms, including over 55,000 classes, 198,000 individuals, and 22,000 relations (predicates and functions). These terms are linked by 3.3 million assertions. The Cyc KB operates in conjunction with an inference engine, with deductive and abductive capabilities. The inference engine is composed of more than 800 special-purpose reasoning modules, each of which handles a very specific type of query. These modules range in complexity from the very simple (special index-lookup code for a specific predicate) to the extremely complex (modules for temporal reasoning). The modules are interlinked via a blackboard system, which allows very general modules to handles cases for which more specific support does not exist; the final fall back is a general theorem prover. This architecture allows for the addition of new modules as needed.

An important aspect in the design of the system comes from the design of the ontology. Specifically, because of the explosive growth of search spaces during deductive inferences with transformation, the ontology needs to be designed such that it can make inference more efficient. The choice not to represent classes with unary predicates is one illustration of this. Another significant illustration of this point is the inclusion of definitional assertions on the elements of the ontology and the enforcement of the semantics represented by those assertions at different stages within the system. The definitional assertions for predicates are used to constrain the values that the predicate's arguments can take (I will refer to these assertions content as "semantic meta-knowledge"). During the knowledge entry process, this enables the system to exclude assertions as "semantically ill-formed" and, thus, ensure the integrity of the content of the KB. Thus, if #$biologicalMother can hold only between an animal and a female animal and female animal and integer are disjoint classes (i.e., they do not and cannot have any common instances), then an attempt to enter into the KB that someone's mother is the number 12 would be rejected.

By preserving the integrity of the knowledge entered into the KB, these semantic restrictions maintain the correctness of the answers derived through deductive inference and spare inference the cost of checking for the satisfaction of constraints.

Another application of this knowledge is with abductive inference. Cyc implements abduction through hypothesizing entities or relations such that the hypothesized facts satisfy the antecedents of deductive implication rules. The inference engine uses this knowledge about the meaning and proper usage of predicates to reject hypotheses that could never be true. Thus, it will not hypothesize that an office building was destroyed because it was a meteor that burned up upon entering the atmosphere, because no office building is a meteorite.

As a first step toward using the knowledge in the Cyc ontology and KB to enhance machine learning, we will be investigating the use of semantic meta-knowledge to provide inductive biases to ILP systems. The next two sections of this paper discuss the declarative representation of bias used by ALEPH, and McCreath and Sharma's [5] algorithm for learning modes and types from data. The remainder of the paper will provide more detail on one form of semantic meta-knowledge for predicates, discuss how to convert them into inductive biases, and discuss additional approaches for handling more complex cases.

3 The Declarative Representation of Bias in ALEPH

ALEPH is an implementation of the inverse entailment algorithm underlying the Progol system (Muggleton [8]). We are interested in this formalism because it combines mode and type information and, so, offers a scheme that could be used among different ILP systems. The system represents mode and type biases using the predicate `mode/2`.[2] The general form of these clauses is:

`mode(RecallNumber,pred(`$\texttt{ModeType}_1$`,…,`$\texttt{ModeType}_n$`).`

`RecallNumber` is either the maximum number of successful calls to the predicate that appears in the second argument position, or an asterisk, meaning the predicate has unbounded indeterminacy. The number of instances of $\texttt{ModeType}_i$ equals the arity of the predicate `pred`. If these mode types are simple, the expressions that will replace $\texttt{ModeType}_i$ begin with a symbol designating the mode of the variables that will fill that argument position, followed by a name for the type constraint on that argument. In ALEPH, input variables are designated with a '+', output variables are designated with a '-', and constant arguments are designated with a '#.'[3] For example, if the predicate `addition/3` has all of its arguments constrained to integers and an expression of the form `addition(X,Y,Z)` means that `Z` is the sum of `X` and `Y`, then the following is an appropriate mode declaration for the predicate:

```
mode(*,addition(+integer, +integer,-integer)).
```

This clause states that `addition/3` has unbounded indeterminacy, that all of its arguments take instances of the same type, and that the first two arguments are input

[2] This following description of ALEPH is based on the documentation available at: http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph.html

[3] This paper will not discuss the creation of constants, but that is an important piece of future research.

variables while the third argument is an output variable. It is important to note that the names given to the types are not used for type reasoning by ALEPH. So, no special significance attaches to the names used in the `mode/2` statements.

The mode settings contained in `mode/2` clauses are used by ALEPH to constrain the set of hypotheses, by shaping the construction of the bottom clause from which the algorithm generalizes. The modes constrain the hypotheses' variables in the following ways:

- An input variable of a given type in a body literal must appear as an input variable of the same type in the head literal or as an output variable of the same type in an earlier body literal of that clause.
- An output variable of a given type in the head literal must appear as an output variable of the same type in some body literal of that clause.
- Any predicate's arguments declared to be a constant of a given type, must take a ground instance of that type as a value in that argument.

The user need not program the mode and type information. ALEPH includes an implementation of McCreath and Sharma's (1995) algorithm for inferring mode and type constraints from the background knowledge and examples.

4 McCreath and Sharma's Algorithm: Bias Induction from Data

4.1 The Algorithm

McCreath and Sharma's [5] algorithm aims to induce meta-knowledge from only the data given to the system. For determining mode biases, the algorithm initially assumes that all possible combinations of modes for a predicates' arguments are valid. It then works through the available data to find counter-examples. When a counter-example is found, the contradicted modes are eliminated. Counter-examples are based on the assumption that modes represent functional dependencies. So, a counter-example would involve the values for a set of input arguments being matched to different values for an associated output argument. For example, this algorithm would originally assign the binary predicate `p/2` the set of possible modes: `p(+,+)`, `p(+,-)`, `p(-,+)` and `p(-,-)`. If `p(a,b)` and `p(a,c)` were positive examples, then the mode setting `p(+,-)` would be eliminated from the set because the examples contradict the claim that the value of the second argument is a function of the first argument.

For determining types, McCreath and Sharma's algorithm begins with the assumption that every argument of every predicate in the language of the learning problem is constrained to a different type. As the algorithm processes the data, if the same value appears in two different argument positions, then the algorithm redistributes the type constraints and assigns the same type to the argument positions that shared that value. For example, if we have two binary predicates `p/2` and `r/2`, then the algorithm would initially assign every argument a different type constraint: `p(type1,type2)` and `r(type3,type4)`. If `p(a,a)` and `r(a,b)` appeared within the examples, then the types would be modified to show that the same type can

appear in both of `p/2`'s arguments and in the first argument of `r/2`. So, the revised types would be `p(type1,type1)` and `r(type1,type4)`.

4.2 Weaknesses of the Algorithm

With any form of learning, there is the possibility of error; but there are two types of error to which McCreath and Sharma's original algorithm and a modified version implemented within ALEPH are susceptible. Both relate to type constraints. Excluding available type constraints reduces efficiency, while including too many type constrains puts too much of a restriction on the search space and, so, causes the system to miss legitimate hypotheses.

McCreath and Sharma's basic algorithm is susceptible to including too few type restrictions. For instance, consider the following representation of a section of a family tree.

```
father(abe,bob).
mother(abe,carol).
father(bob,david).
father(carol,fred).
```

In this example, Bob and Carol both appear in the first argument positions of `father/2` clauses. This causes the types on the second argument positions of `mother/2` and `father/2` to be merged, even though they should be constrained to disjoint types: female and male animals, respectively. So, in this situation, there would be only one type for all of the argument positions.

ALEPH includes an alternative implementation of the algorithm that allows for the merging of types between argument positions based on the degree of overlap in the values for two different positions, where the degree of overlap is the proportion of values of one position that are also values for the second position. This approach faces the problem of not recognizing that argument positions can be merged. For instance, in the above example, because there is only one value that appears in both the second argument of mother and the first argument of father, a sufficiently high overlap threshold between those arguments would prevent them from being linked. Thus hypotheses that link mothers to their fathers could be missed.

5 Semantic Meta-knowledge in Cyc

5.1 Basic Meta-knowledge for Predicates

A first step in the addition of a new predicate to the Cyc ontology is the creation of definitional assertions that express the predicate's meaning by setting constraints on its applicability. For example, the predicate `#$hasHeadquartersInRegion` relates an organization to the geographical region where that organization's headquarters are located. If `#$BMWInc` denotes the Bayerische Motoren Werke corporation and `#$CityOfMunich` denotes the city of Munich, Germany one could assert:

```
(#$hasHeadquartersInRegion #$BMWInc #$CityOfMunich)
```

To enforce the semantics of this predicate, a number of other assertions that constrain #$hasHeadquartersInRegion would be used. For example, #$arg1Isa and #$arg2Isa are, respectively, used to state that the first and second argument positions of a particular predicate must be instances of particular collections.[4] In order to express that the first argument of #$hasHeadquartersInRegion must be an organization and second argument must be a geographical area, the following assertions would be made:

```
(#$arg1Isa #$hasHeadquartersInRegion #$Organization)
(#$arg2Isa #$hasHeadquartersInRegion
#$GeographicalRegion)
```

The effect of these two statements is that if a knowledge engineer tries to enter a new assertion using the predicate #$hasHeadquartersInRegion and the value of one of its arguments can be proved to not be an instance of the required type, then the semantic validation process would reject the assertion.

Many predicates in the Cyc ontology take collections as values. The predicate #$argIsa can be used to state that a value must be a collection. Additional assertions can further specify the collection. First, the predicate #$argGenl can be used to state that the value must be a sub-collection of some collection. Alternatively, instead of #$Collection, #$argIsa could refer to a second-order collection – a collection whose instances are collections. #$PersonTypeByPositionInOrg is a second-order collection whose instances are types of occupation that refer to organizations (e.g., #$PrimeMinister, #$ChiefExecutiveOffice). The predicate #$personHasPositionInOrg relates a person to a type of occupation and an organization. The following definitional assertions hold of its second argument:

```
(#$arg2Isa #$positionOfPersonInOrg
        #$PersonTypeByPositionInOrg)
(#$arg2Genl #$positionOfPersonInOrg    #$Person)
```

So, although an instance of the collection #$PrimeMinister (e.g., Tony Blair) could fill the first argument of a #$positionOfPersonInOrg assertion, the appearance of the same value in the second argument position would not be semantically well-formed because the collection of individuals is disjoint with the collection of occupation types, #$PersonTypeByPositionInOrg.

5.2 Representing the Determination

The first step is the process is the representation of the problem, or determination. Specifically, there is the representation of the target predicate (i.e. the predicate for

[4] The Web Ontology Language, OWL, based on its inheritance of certain portions of the Resource Description Framework, includes similar vocabulary for defining the properties of predicates. The Cyc ontology includes a larger vocabulary with that function, but the particular process described here can be applied to OWL ontologies as well.

the head of learned clauses) and the set of predicates for the background knowledge (i.e., the predicates that could appear in the bodies of learned clauses). The Cyc ontology uses the function `#$ILPDeterminationFn` for denoting determinations. It is a binary function whose first argument position is the target predicate and whose second argument is the list of background predicates. In order for the target predicate to be considered a possible body predicate, it would need to be included in the list of predicates as well.

For example, the following expression represents a determination that targets the predicate `#$hasHeadquartersInRegion` with the possible body predicates: `#$residesInRegion`, `#$seniorExecutives`, `#$importantCompany`, and `#$positionOfPersonInOrg`:

```
(#$ILPDeterminationFn
          #$hasHeadquartersInRegion
          (#$TheList #$residesInRegion
          #$seniorExecutives #$importantCompany
          #$positionOfPersonInOrg))
```

For simplicity's sake, during the remainder of this section the term `#$Determination01` will be used as a name for the determination denoted by the above expression.

5.3 Deriving Types

Given a specific determination, Cyc uses its semantic meta-knowledge and its type hierarchy to determine a more compact representation of the relevant type constraints. This more compact representation allows us to communicate information to ILP engines like ALEPH that do not do type reasoning when using inductive biases. The goal is to try to capture all and only the appropriate linkages among arguments, while minimizing the size of the representation. Because of the size of the Cyc ontology and the different representational tasks of different predicates, they have type constraints at different levels of generality. The basic strategy implemented here is to look for subsumption relations among argument constraints and then to use only the most specific types that are used as constraints.

The first step in the process is to generate the set of collections that constrain the argument positions for the predicates in the determination. The set of these collections for `#$Determintion01` is:

```
{#$Animal, #$PersonTypeByPositionInOrg #$Person,
#$Organization, #$CommercialOrganization,
#$GeographicalRegion}
```

The next step is to reduce the set by eliminating any collections that have at least one sub-collection that is also element of the set. `#$Person` is a sub-collection of `#$Animal` and `#$CommercialOrganization` is a sub-collection of `#$Organization`. So, `#$Organization` and `#$Animal` would be eliminated to produce the new set:

```
{#$Person, #$PersonTypeByPositionInOrg
#$CommercialOrganization, #$GeographicalRegion}
```

For the remainder of this paper, any reference to a "relativized type" should be taken to refer to an element of this reduced set. Given the original semantic meta-knowledge about the predicates, knowledge of the subsumption and disjointness of collections, a set of rules are invoked that conclude to the predicate `#$argIsaWRTSpec`. ("argument is an instance of type with regard to determination"). This predicate is used to represent more focused type constraints for a predicate within the context of a given determination. For example, relative to the determination above, Cyc would infer

```
(#$argIsaWRTSpec
#$hasHeadquartersInRegion 1 #$CommercialOrganization
#$Determination01)
```

This statement states that, relative to `#$Determination01`, the first argument of the predicate `#$hasHeadquartersInRegion` should be treated as if it were constrained to `#$CommercialOrganization`. In cases where the reduced set contains several collections that are subsumed by the predicate's original argument-constraining collection, they are all represented with `#$argIsaWRTSpec` assertions for that predicate's argument.

5.4 Deriving Modes

For certain predicates, modes are readily definable based on the content of the underlying relation. For example, in the case of a predicate that relates a list to one of its members, a mode that requires the list value to be provided (i.e., an input) and the member value to be derived (i.e., an output) is more efficient than the inverse would be since the member value is a member of an unlimited number of possible lists. For the vast majority of predicates in the Cyc ontology, a similar judgment is not as readily available. The issue is magnified by the fact that the choice of mode settings can greatly influence the behavior of the ILP system.

As Cyc is an engineering project, the choice of how to proceed with establishing those connections occurs in the context of improving Cyc's ability to answer questions. Specific applications of Cyc will likely focus on a central set of queries that share a common set of predicates, most of those predicates will be used to describe a small set of types of entities. Assuming that induction will be geared to improving inference with regard to a particular topic, the current strategy for generating modes is to focus on one argument of the target predicate. We expect that this focal argument will be bound during a query. In practice, this binding would be the entity that is the topic of the question. So, if we wanted to know information about commercial organizations, we would ask the following sort of query:

```
(#$hasHeadquartersInRegion #$BMWInc ?PLACE)
```

Given that the focal type for these queries is `#$Organization` and that the first argument of `#$hasHeadquartersInRegion` is the only one constrained to

#$Organization, we would include the following assertion to specify that the organization argument is focal for this determination:

```
(#$focalArgumentOfDeterminationTarget
#$Determination01 1)
```

Once a focal argument has been designated, the following rules are used to determine the modes for the determination's predicates:

- The focal argument of the target predicate has an input mode.
- If the target predicate has an input argument of relativized type T, and a predicate (either target or background) also has an argument of relativized type T, then that second argument has an input mode.
- If the target predicate has an input argument of relativized type T_1, a predicate (either target or body) has an argument of relativized type T_1, and T_1 and T_2 are disjoint collections, then that second argument has an output mode. [5]

This algorithm utilizes the user-provided information of the focal argument for the target predicate in conjunction with semantic knowledge regarding all of the predicates in the determination to derive modes for each argument.

In CycL, the process concludes with the derivation of assertions represent the combination of the relativized type for a predicate's arguments and that argument's mode. The predicates #$argIsaInputModeForDetermination and #$argIsaOutputModeForDetermination are used for this task. In the case of #$Determination01, we have assumed a focal argument that is constrained to organizations. Since organizations are disjoint with geographical regions, the system would draw the following conclusions for the predicate #$importantCompany, which relates a commercial organization to a geographical region in which it plays a significant economic role:

```
(#$argIsaInputModeForDetermination
          #$importantCompany
          #$CommercialOrganization 1
          #$Determination01)

(#$argIsaOutputModeForDetermination
          #$importantCompany
          #$GeographicalRegion 2
          #$Determination01)
```

The first argument is an input mode because it shares the same relativized type as the focal argument, #$CommercialOrganization. The second argument is an output mode because its relativized type, #$GeographicalRegion, is disjoint with the focal argument's relativized type.

[5] This is a rough reconstruction of the general pattern found in the data sets found in the archive at: http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/applications.html. The development of more sophisticated algorithms is a topic for future research.

5.5 More Complex Cases

The method described above is most efficient in cases where the set of type constraints all involve disjointness and subsumption relationships. First, the subsumption allows us to minimize the number of collections that need to be considered. So, we can send out a more compact representation to the ILP engine. Further, when two collections are disjoint, then there should be no relationship between two arguments that are constrained by those collections. However, for any pair of collections that are chosen from the ontology, a significant chance exists that they are neither disjoint nor does one subsume the other. As of March 2005, the probability that two randomly selected collections from the Cyc KB are orthogonal is 0.35. While such 'orthogonal collections' are a minority in the set of collections used for semantic constraints on predicates, they do occur. For example, the second argument of `#$father` is constrained to `#$MaleAnimal`, while the first argument of `#$rulerInDynasty` is constrained to `#$Monarch-HeadOfState`. Neither of these collections subsumes the other and they share instances. Depending on the focal argument of the determination, the rules described might not return modes for such cases. There are four solutions that present themselves for handling cases where this situation is violated.

First, one could introduce a framework that allows for type conversions of individuals within the rules that govern the use of modes. Such conversions would be made in the case of individuals that are instances of multiple collections. A second strategy is the introduction of more general collections that subsume the orthogonal collections to link the different argument. Third, a more specific collection could be introduced that represents the intersection of the various orthogonal collections. Finally, temporally created specialized predicates could be introduced when communicating the background knowledge to ALEPH. The new predicates would be specialized in the sense that their argument constraints be the intersection of the orthogonal collections. This would establish subsumption relations and make them compatible with the simple algorithm described above.

First, the "Casting" strategy is intended for cases where predicates are constrained by orthogonal collections and they have values that are instances of those orthogonal classes (e.g. a father who is a ruler in a dynasty). The strategy focuses on type constraints of predicates, but also takes into account the fact that some entities will be instances of more than one collection. So, if an argument is constrained to a particular type that is orthogonal to another predicates' constraining type and the value of that argument is also an instance of the orthogonal collection, then that value should be used with the other predicate. The principle cost of this strategy is a dramatic increase in the number of predicates in relation to the number of individuals that are instances of multiple collections. In addition, since type conversion has to be bi-directional, there is the potential of catching ALEPH in a conceptual loop, as it oscillates back and forth between two facets of a term.

Second, the "Type Generalization" strategy walks up the type hierarchy to find the least general collection that subsumes the orthogonal collections. From a strictly semantic perspective, the use of the more general types violates the meaning of the predicate. This violation translates into a decrease in the utility of type knowledge. Since fewer distinctions are being made, there is a weaker partitioning of the search

space. In the degenerate case, all type knowledge is lost if the least general collection remains general enough to subsume all the argument constraints of all the predicates in the determination.

Third, the "Type Specialization" strategy introduces more specific types that allow the system to maintain the semantic integrity of the predicates and the benefits that follow from the splitting up of the search space. However, this strategy requires the generation of significantly more types and a proportionately larger group of mode statements that properly connect them to the predicates.

Fourth, the "Predicate Specialization" strategy introduces more specific predicates. It requires creating more specialized predicates whose argument positions have significantly tighter constraints and then distributing the original data to the new predicates based on the types of their values. Here, the problem of having multiple linkages to the same argument positions is eliminated because the examples will be partitioned based on the new predicates' tighter semantics. However, there are two problems with the approach. First, the strategy increases the likelihood of the system returning over-fitted rules. If the invented predicate has a very small extent, the likelihood increases that it will satisfy some of the positive examples even though the more general predicate from which it was derived would not. Alternatively, if the system's evaluation function has a relatively high threshold for acceptability (e.g., its minimum number of positive examples covered), the split might be such that hypotheses with the new predicate falls beneath that threshold while a rule formed with the original predicate would be above it.

6 Conclusions and Future Research

The next and most important stage in this work is the empirical investigation of the utility of this strategy for ILP systems. Existing data sets need to be ontologized such that a system like ALEPH can be tested with ontologically derived settings versus hand-generated settings and the settings provided by other mode learning algorithms.

In addition, our presentation has focused on only a subset of the types of semantic meta-knowledge that are available within the Cyc. Cyc contains a large amount of unexploited meta-knowledge, such as that expressed by `#$interArgReln` and `#$interArgIsa`. The former predicate is used to say that a particular binary relation must hold between the values of some predicate. The latter predicate dictates that when an instance of one collection appears in one argument of a predicate, then an instance of a specified collection must appear in some other argument (e.g., a human child can only have a human as a parent).

The content of the Cyc KB itself can also serve as a resource for generating inductive biases. It contains a large body of rules expressed as either implications or as ground assertions that can be expanded to implications via templates. These rules could be used to divert the search away from known rules or hypotheses that are entailed by existing rules. Preliminary work has been done to convert rules into formats that use ALEPH's pruning mechanism toward this end.

Finally, we have introduced only a simple methodology for generating mode settings based on type constraints. More sophisticated strategies for deriving modes from the focal type and the relationships among the type constraints of the

determination should become available with further research. Given that the rules for generating modes are declaratively represented, representing the modes themselves and rules for interpreting them could lead to further rule-learning improvements.

References

1. Tausend, B.: Biases and their Effects in Inductive Logic Programming. In: F. Bergadano and L. De Raedt (eds.): Proceedings of the 7th European Conference on Machine Learning. Lecture Notes in Artificial Intelligence, Vol. 784. Springer-Verlag, Berlin Heidelberg New York (1994) 431–434.
2. Tausend, B.: Representing Biases for Inductive Logic Programming. In: F. Bergadano and L. De Raedt, (eds.): Proceedings of the 7th European Conference on Machine Learning. Lecture Notes in Artificial Intelligence, Vol. 784. Springer-Verlag, Berlin Heidelberg New York (1994) 427–430.
3. Nedellec, C., Rouveirol, C., Ade,H., Bergadano, F., Tausend, B.: Declarative Bias in ILP. In: L. De Raedt (ed.): Advances in Inductive Logic Programming. IOS Press, Amsterdam (1996) 82-103.
4. McCreath, E.: Induction in First Order Logic from Noisy Training Examples and Fixed Example Set Sizes. PhD Thesis, University Of New South Wales (1999).
5. McCreath, E. and Sharma, A.: Extraction of Meta-Knowledge to Restrict the Hypothesis Space for ILP Systems. Yao, X. (ed): Eighth Australian Joint Conference on Artificial Intelligence. World Scientific Publishing, Singapore (1995) 75-82.
6. Di Mauro, N., Esposito, F., Ferilli, S. and Basile, T.M.A.: An Algorithm for Incremental Mode Induction. In Orchard, B., Yang, C., Moonis, A. (eds.): Innovations in Applied Artificial Intelligence: 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE 2004. Lecture Notes in Computer Science, Vol. 3029. Springer-Verlag, Berlin Heidelberg New York (2004) 512-522.
7. Lenat, D. and Guha, R.: Building Large Knowledge Based Systems: Representations and Inference in the Cyc Project. Addison-Wesley, Reading (1989).
8. Muggleton, S.H.: Inverse Entailment and Progol. *New Generation Computing* **13** (1995): 245-286.