

Using Access Paths to Guide Inference with Conceptual Graphs

Peter Clark¹ and Bruce Porter²

¹ The Boeing Company, PO Box 3707, Seattle, WA 98124
(clarkp@redwood.rt.cs.boeing.com)

² University of Texas at Austin, TX 78712 (porter@cs.utexas.edu)

Abstract. Conceptual Graphs (CGs) are a natural and intuitive notation for expressing first-order logic statements. However, the task of performing inference with a large-scale CG knowledge base remains largely unexplored. Although basic inference operators are defined for CGs, few methods are available for guiding their application during automated reasoning. Given the expressive power of CGs, this can result in inference being intractable.

In this paper we show how a method used elsewhere for achieving tractability — namely the use of *access paths* — can be applied to conceptual graphs. Access paths add to CGs domain-specific information that guides inference by specifying preferred chains of subgoals for each inference goal (and hence, other chains will not be tried). This approach trades logical completeness for focussed inference, and allows incompleteness to be introduced in a controlled way (through the knowledge engineer's choice of which access paths to attach to CGs). The result of this work is an inference algorithm for CGs that significantly improves the efficiency of reasoning.

1 Introduction

Consider asking a large-scale Conceptual Graph (CG) knowledge-base the question “What is the age of the American president?” Answers to this question may be determined in many ways by applying CG inference rules. Some inference chains are more likely to be fruitful than others. For example, a fruitful chain might be: “The age of a person is probably (approximately) the age of his/her spouse”, hence “Is the president married?” and “What is the age of his spouse?”. A less fruitful chain might be: “The age of a person is the age he/she declared on his/her last job application”, hence “What was the last job the president applied for?”, hence “A person's last job probably matches his/her interests”, hence “What are the president's interests?”, hence “A person's interests probably match his/her friends' interests”, hence “Who are the president's friends?”, etc. Clearly, to answer questions in a timely fashion and to cope with the inherent intractability of inference with expressive languages such as CGs, some mechanism is needed to guide the application of the basic inference rules. Although conceptual graph research has significantly improved the efficiency of basic inference operations (eg. performing joins), there is still a need for methods that guide their application so that questions can be efficiently answered from large CG knowledge-bases.

In this paper, we apply ideas from *Access-Limited Logic* [1] to Conceptual Graphs, as a means of guiding CG inference. Access-Limited Logic specifies *access paths* that (1) relate together the concepts in a knowledge-base, and (2) constrain inference to follow only those paths when answering queries that require navigating the knowledge-base. Although this introduces logical incompleteness, it does so in a controlled way, as the knowledge engineer chooses which access paths to include in the knowledge-base. In addition, access-limited logic retains the property of “Socratic Completeness”, which guarantees that no consequence of the knowledge base is inherently unreachable – there will always be some sequence of queries which can be issued to the KB allowing any logical consequence of it to be inferred. Our experience in applying this approach to a large-scale CG knowledge-base is that suitable paths can generally be determined from domain knowledge and encoded within a CG framework. The result of this approach is that complex chains of reasoning become possible with conceptual graphs because the inference process is focused in fruitful ways.

We have used the representation language described here, which adds access paths to standard CGs, to build a large knowledge-base about plant biology [2] and a smaller knowledge-base about distributed computing [3]. We have fully implemented the inference algorithm presented here, and used it in conjunction with these knowledge bases [4].

The paper is organized as follows. Section 2 outlines the CG semantics used in this work. Section 3 describes access paths and how they can be used to guide inference, and Section 4 presents an inference algorithm based on this approach. Sections 5 and 6 discuss some of the benefits and limitations of this approach.

2 Inference with Conceptual Graphs

2.1 The CG Knowledge Base

We consider a CG knowledge-base (KB) to contain two basic types of representational structures, namely *schemata* and *type definitions*, as illustrated in Figure 1. Intuitively, a schema for a concept C denotes “contingent facts” about C , namely those facts that are implied by membership in C , but which are not sufficient to conclude membership. This corresponds to uni-directional logical implication. A type definition for concept C , in contrast, denotes the “definitional properties” of C , ie. those facts that are both implied by membership in C and (together) are sufficient to conclude membership in C for some individual. A type definition corresponds to bi-directional logical implication. In terms of Description Logics (eg. [6]), type definitions express terminological (TBox) knowledge, while schemata express assertional (ABox) knowledge. In a knowledge-base, a concept will typically have both a type definition (describing its definitional properties) and a schema (describing additional implied properties). In addition, the KB contains a type lattice (‘hierarchy’) asserting generalization/specialization relationships among types.

2.2 Inference Rules

There are different, interchangeable vocabularies with which CG inference can be described. At the most primitive level, (Sowa’s CG adaptation of) Peirce’s beta

<p><u>Conceptual Graph:</u> schema for $concept(x)$ is <i>graph of preds using x, y_1, \dots, y_n</i></p>	<p><u>Semantics:</u> $\forall x \text{ concept}(x) \rightarrow$ $\exists y_1, \dots, y_n \text{ set of preds using } x, y_1, \dots, y_n$</p>
<p>type $concept(x)$ is <i>graph of preds using x, y_1, \dots, y_n</i></p>	<p>$\forall x \text{ concept}(x) \leftrightarrow$ $\exists y_1, \dots, y_n \text{ set of preds using } x, y_1, \dots, y_n$</p>
<p>For example:</p>	
<p><u>Conceptual Graph:</u> schema for PERSON(x) is [PERSON: *x] - (PART) \rightarrow [HEAD] .</p>	<p><u>Semantics:</u> $\forall x \text{ person}(x) \rightarrow$ $\exists y \text{ part}(x, y) \wedge \text{head}(y)$</p>
<p>type RED-WINE(x) is [WINE: *x] - (COLOR) \rightarrow [RED] .</p>	<p>$\forall x \text{ red-wine}(x) \leftrightarrow$ $\exists y \text{ wine}(x) \wedge \text{color}(x, y) \wedge \text{red}(y)$</p>

Fig. 1. Semantics used for CG schemata and type definitions.

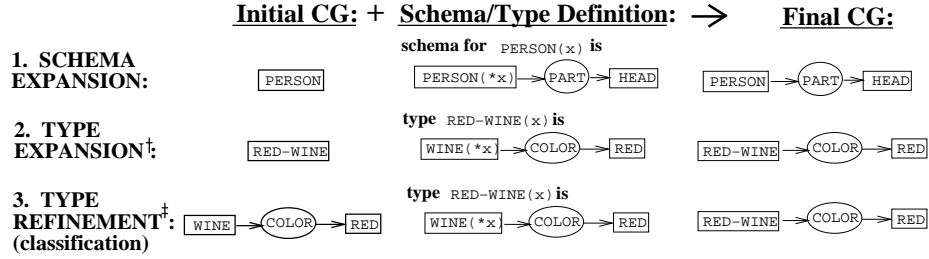
rules provide a set of graph rewriting rules for CGs, equivalent in power to the rules of first-order predicate calculus [5, page 154]. These rules can be thought of as the “assembly code” for CG inference, in that they are building blocks with which other inference rules (“derived rules” [5, page 151]) can be described. For example, modus ponens is equivalent to applying a particular sequence of four alpha/beta rules. This approach is used in other CG implementations (eg. Prolog+CG [7]), and also here.

Type definitions and schemata express three types of inference rules, corresponding to the three directions of implication in Figure 1:

1. concept membership implies facts (schema expansion)
2. concept membership implies facts (type expansion)
3. facts imply concept membership (type refinement)

These three rules form the basic inference rules for our knowledge-base. They are derived rules, as they can each be implemented as a sequence of Peirce’s beta rules. However, for simplicity we treat and implement them as primitives.

In graph theoretic terms, a **schema expansion** is a minimal join of a schema to a conceptual graph, and is equivalent to applying the inference rule which the schema represents. Similarly, we define a **type expansion** as a minimal join of a type definition to a concept in a conceptual graph, which is equivalent to applying the type definition’s inference rule in the forward direction (“concept membership implies facts”). Finally, we define a **type refinement** as the recognition that a type’s definition is satisfied by a concept in a conceptual graph (ie. the definition subsumes the graph at that node), and the resulting specialization of that node’s type in the graph to be that in the type definition. A type refinement is equivalent to type contraction, but without detaching items from the CG, and corresponds to applying the type definition’s inference rule in the



[†] This definition slightly deviates from that in [5], in that RED-WINE in the initial CG is *not* replaced by WINE in the final graph, but instead is retained. This enables types/schemas for RED-WINE and its generalizations to be located for future expansions.

[‡] Type refinement is equivalent to type contraction [5, page 108] without detaching items from the CG.

Fig. 2. The three basic CG inference operators used.

backward direction (“facts imply concept membership”). Expansions correspond to the *elaboration* of initial knowledge, and type refinements correspond to the *classification* of concepts in a conceptual graph. These are illustrated in Figure 2.

2.3 Cardinality Assumptions

In the literature, relations are normally assumed to be multivalued (ie. many-to-many) unless otherwise specified. To denote functional relations (ie. one whose second argument is unique, given the first), the ‘@1’ notation is normally used (eg. to denote that a person has just one father, the PERSON schema would include [PERSON] → (FATHER) → [MAN:@1]). Functional relations are common in knowledge-bases, and also have particular significance when performing minimal joins: minimal joins will only merge relations which are functional (ie. will only assume coreferentiality when logically implied), and hence their use is essential if a schema/type definition is intended to imply extra properties of objects in a CG (rather than assume objects in the schema are distinct from those in the CG). Thus, as a notational convenience in this paper and our implementation, we assume that relations *are* functional unless otherwise specified (using a @* annotation). As a result, minimal joins will by default merge matching relations together, unless annotated otherwise.

3 The Inference Task and the Use of Access Paths

3.1 Inference with CGs

Our goal is to use a CG knowledge base to answer queries. A query comprises a set of assertions (eg. “there exists a table”) and a question (“what is it made of?”). We call the assertions the ‘scenario’, and represent the scenario as a CG whose nodes are all instances. The scenario is a temporary CG, built just for answering the question, and to be elaborated until it contains the answer. We can thus describe the inference problem as follows:

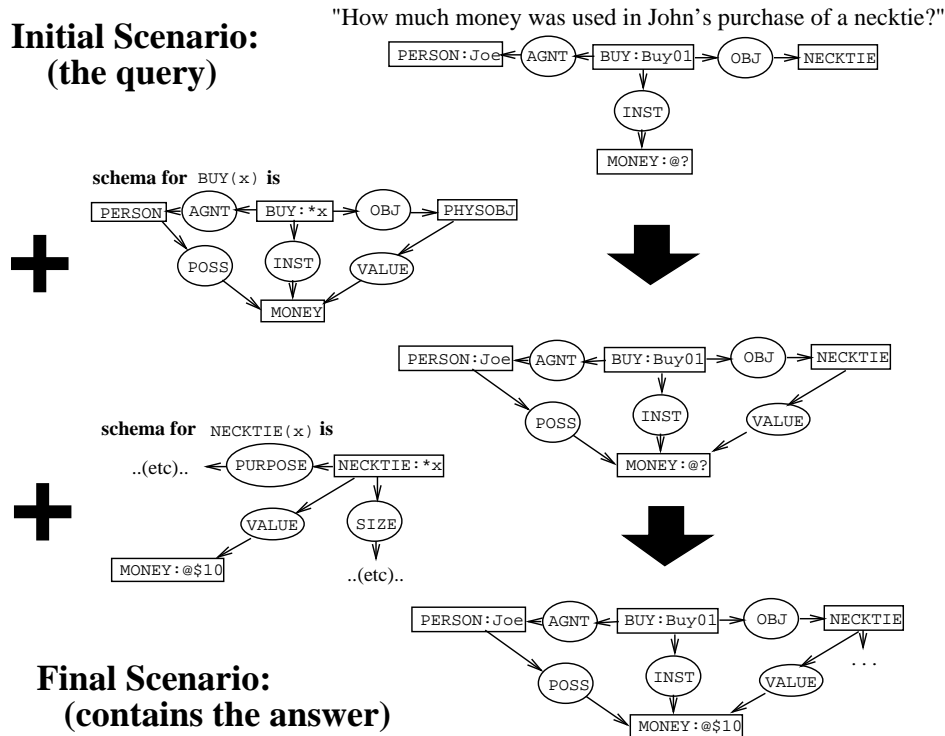


Fig. 3. Inference with schemata only: Expanding the Buy01 and NECKTIE nodes in the initial CG produces an answer to the query.

- **GIVEN:**
 - A CG KB of type definitions and schemata, and a type lattice
 - A query, comprising assertions (the initial scenario CG) and an unknown variable (the target)
- **FIND:**
 - A sequence of expansions/refinements to the initial scenario CG which results in it containing a value for the target.

3.2 The Search Problem

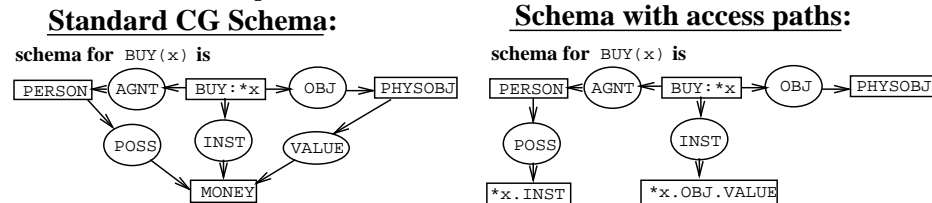
A simple example of this inference task is shown in Figure 3, an adaptation of [5, page 110]. (We ignore issues related to temporal reasoning, as they are not our concern here). In this example, the initial CG describes "Joe buying a necktie", and the query asks "How much money was used in the purchase?" In the scenario graph, there are three nodes that could be expanded by joining each node's schema (from the KB) to the initial graph. (We assume that schema are defined for each node.) Although in this case just two schema expansions will answer the query (as shown in Figure 3), many more could be made; most of these alternatives fail to answer the query. For example, suppose the first

expansion joined the BUY(x) schema from the KB with Buy01. As this fails to answer the query, further expansions are needed, and (at least) three are possible: expand PERSON, NECKTIE, or Buy01 (using a schema from some other generalization of BUY). However, there is no information available to suggest which should be tried first: all three relate to the node of interest (MONEY) in structurally identical ways. If PERSON (say) is expanded, then again no answer is found and more concepts will be added to the initial CG, providing yet more choices for the next expansion, and so on. In general, the space of sequences of expansions and refinements is too large to search exhaustively.

3.3 Access Paths

Of course, this problem is not specific to CGs, but occurs whenever a logically complete inference procedure is applied to an expressive representation language. One solution to this problem is to significantly weaken the language’s expressive power. This approach is pursued by many within the Description Logic community (eg. [6]). However, the language restrictions required to guarantee tractability are severe and can limit the language’s applicability [8]. The alternative we have chosen is to restrict the inference process, thus achieving tractability at the expense of completeness. The challenge is to adopt restrictions that, in practice, have minimal effect on question-answering ability.

The inference control method we present for CGs is based on “access limitation”, which is used elsewhere in the knowledge representation community [9]. Access limitation is based on the use of *access paths*, which state how a value can be computed via a chain of inferences. In the context of Conceptual Graphs, an access path describes how the value of a CG node can be determined by traversing a path in the graph. Incorporating access paths into CGs requires a small extension to the CG formalism, whereby a variable at a CG node can be replaced with a path, *at the end of which the value of that variable may be found*. To the inference engine, the path describes a sequence of subgoals that may result in a variable’s value being found. As an example of access paths, consider the following schema for BUY(x):



where $*x.OBJ.VALUE$ denotes the path $[*x]-(OBJ) \rightarrow [*]-(VALUE) \rightarrow [*]$, or equivalently in logic notation:

$$obj(Self, y), value(y, z)$$

(where *Self* refers to the instance of BUY under consideration). This particular path provides guidance for computing the INSTRUMENT of the BUY. Specifically, it instructs the inference engine to first find the OBJECT bought, then find its VALUE (thus, it indirectly instructs the inference engine *not* to consider (say) the PERSON node further for computing this value). In this way, paths encode a

preferred sequence of computations for computing the identity of one concept from others, and this knowledge can be used by the inference engine to reduce the search to answer a query. Paths may point to concepts which themselves are described by paths, and hence complex chains of inference may be required to find the concept(s) to which a path refers.

More formally, we define an access path as a chain of binary predicates P_i linking some individual $X0$ to other individuals x_n , such that the second argument of P_i is the same as the first argument of P_{i+1} :

$$P_1(X0, x_1), P_2(x_1, x_2), \dots, P_n(x_{n-1}, x_n)$$

where the x_i 's are free variables. (This is a slightly more restricted definition than in [9], but one suitable for conceptual graphs.) An access path denotes the set S of values for x_n (the last variable in the path) for which there exists at least one value for all the other variables in the path, ie.

$$\forall x_n (x_n \in S \leftrightarrow \exists x_1, \dots, x_{n-1} P_1(X0, x_1) \wedge \dots \wedge P_n(x_{n-1}, x_n))$$

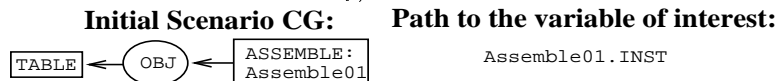
For example, the access path $parent(John, x), sister(x, y)$ denotes "John's parents' sisters". In terms of conceptual graphs, an access path corresponds to the set of paths that start at node $X0$ and traverse arcs labeled P_1, \dots, P_n . For legibility, we write paths using a 'dot' notation of the form $X0.P_1. \dots .P_n$, for example $parent(John, x), sister(x, y)$ would be written $John.parent.sister$.

A chain of predicates like this is very common in knowledge-based systems (and object-oriented programming) and has many different names, including "attribute paths" in CLASSIC [6, page 425], "relation compositions" in LOOM [10], "role chains" in KRYPTON [11, page 421], and "chains" in [12]. The difference here is that chains are used not only to express coreferentiality of concepts, but also to guide inference: Paths do not just denote the concepts they point to, they also give a *sequence of steps by which the identity of those concepts can be computed*, as shown in the next section. It is thus important to retain paths within the CG knowledge-base, rather than "compile them out" into shared variables (as is done, for example, in the language LIFE [13]). Replacing paths with shared variables loses the control information that paths contain.

4 Inference with Access Paths

4.1 An Algorithm for Inference

Inference is performed when a query is issued to the KB, either from a user or an application system. As described in Section 3.1, a query consists of a set of assertions, plus a target variable of interest related to those assertions. In the CG framework, these assertions are provided as an initial 'scenario' CG, and the target is specified as an access path from a node in that graph to the variable of interest. For example, the query "What is needed to assemble a table?" is specified as a scenario, in which there is an assembly (`Assemble01`, say) of a table, and a path `Assemble01.INST` denoting the variable of interest (namely the instrument used in that assembly):



The path starts from a node in the initial graph, but it may reference nodes that are not in the graph; consequently, the graph may need to be expanded to add nodes, in order to find the item(s) which that path points to.

We define **evaluating a path** as the computation of the values which that path denotes. Answering a query is thus the task of evaluating the path, specified in the query. To evaluate a path, it is traversed one arc at a time in the scenario CG. If the CG does not contain the arc to traverse, then the graph is first elaborated by doing a join of some CG in the KB with the current node which the inference algorithm is at in its path traversal. If, at the end of the traversal, the resulting node is itself a path, then the algorithm is called recursively to evaluate that path. Finally, a value will be found and returned. Note that the path guides the algorithm by telling it which arcs in the scenario to traverse, and hence which nodes in the scenario to elaborate (via joins) should an arc be missing. This algorithm is in Figure 4, without the “classify” step being used.

If the scenario is missing an arc that the algorithm wishes to traverse, the algorithm searches up the type hierarchy from the current node N in the traversal, looking for type definitions or schemata describing generalizations of that node, and which contain the missing arc. If one is found, it is joined with node N in the scenario, and hence path following can resume. This algorithm is suitable for a CG KB containing schemata only. However, an extra complication is added if the KB also contains type definitions. Recall from Figure 1 that type definitions allow instances to be classified. Now, it could happen that the currently known type in the scenario for node N is not the most specific possible (as some type definition may imply this instance is of a more specific type). Without adding a classification step to check N is currently classified as specifically as possible, the algorithm’s search up the type hierarchy will start at too general a type, and hence potentially miss some applicable CGs in the KB. For example, if the algorithm is looking for CGs to join with a node N , whose type is ADHESIVE and which (according to the scenario) is being used to affix WOOD, and a NAIL is defined as an ADHESIVE which joins WOOD, then N could first have its type refined from ADHESIVE to NAIL. It is important that such refinements are performed before searching up the type hierarchy, in order to ensure that search starts at the most specific types. To achieve this, we add a ‘classify’ step in the algorithm in Figure 4, which first checks that the instance N is classified as specifically as possible before it is elaborated. Again based on access-limited logic, classification is tractable (but incomplete), as it may call the algorithm recursively to compute properties of the individual being classified (in order to test whether it satisfies a type definition) which itself is a path-guided (hence tractable but incomplete) process.

An additional feature of our implementation is that it performs only *partial joins* of CG schema to the scenario CG. We define a partial join of CG_1 to CG_2 as the join of a *subgraph* of CG_1 to CG_2 . During elaboration, after finding an applicable schema to join to the scenario (step 1 of Elaborate in Figure 4), the algorithm joins only the part of the schema containing the arc of interest with the scenario CG. This part consists of the arc of interest, plus all nodes reachable

```

Procedure evaluate_path(Path = (P1(X0,x1),...,Pn(xn-1,xn)), CG)
  returning values for xn:
  /* GOAL: Evaluate the path left-to-right to find the xn */
  1. For i = 1 to n:
    Compute all values of xi that satisfy Pi(xi-1,xi) by calling:
    xi = evaluate_step(Pi(xi-1,xi), CG)
  2. Return the set of values xn

Procedure evaluate_step(P(X0,y), CG)
  returning values for y:
I: CLASSIFY (type refinement):
  /* GOAL: Try to refine X0's classification, so as to ensure the subsequent
  elaboration step (below) finds all relevant info about X0. */
  1. Find the current most specific class(es) {C1,...,Cn} of X0
  2. For each Cj in {C1,...,Cn}:
    2.1 For each direct specialization SpecC of Cj:
      - Compute whether X0 satisfies the type definition of SpecC (this
      may involve calling evaluate_path() recursively, to query
      for other facts about X0).
      - if X0 satisfies the type definition of SpecC
      - then - refine the class of X0 by replacing Cj with SpecC in CG
      (= type refinement)
      - search for further refinements, by computing whether
      direct specializations of SpecC apply (by recursively
      applying step 2.1 with SpecC instead of Cj).

II: ELABORATE (type/schema expansion):
  /* GOAL: Expand CG at node X0 to include arc(s) P(X0,y). */
  if CG already contains arc(s) P(X0,y)
  then return the value(s) y (no elaboration needed)
  else 1. Search generalizations XGeni of X0 for type definitions
  or schemata with info about P(X0,y), ie. which include the
  arc P(XGeni,Yexpr) in their graphs
  2. For the Yexpr found:
    if Yexpr is a named individual
    then y = Yexpr
    else if Yexpr is a concept (denoting an unnamed individual)
    then create y = a new Skolem constant denoting it in CG
    else if Yexpr is a path
    then compute values for y with evaluate_path(Yexpr,CG)
  3. Add the arc(s) P(X0,y) to CG for all y found (partial join)
  4. Return the value(s) for y.

```

Fig. 4. The full inference algorithm, describing how access paths guide the choice of schema/type expansions/refinements to perform.

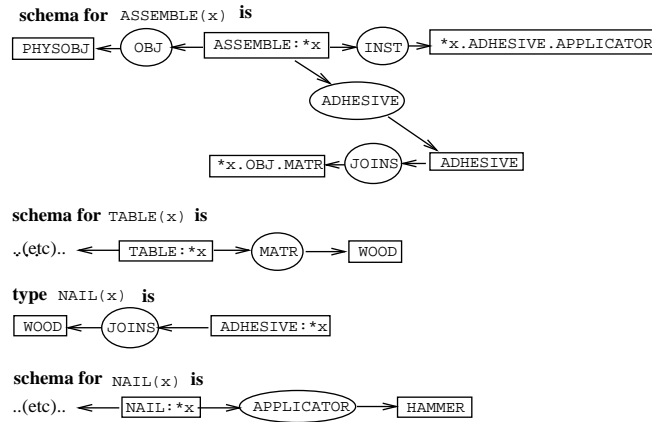


Fig. 5. Part of a CG KB required for answering the query of Section 4.2

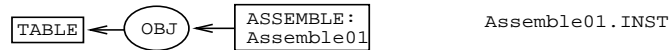
from it in the schema (this will not necessarily be the whole schema). This is done for efficiency, to avoid computing the joins of other parts of the graphs unrelated to answering the current query.

4.2 An Example

We continue with the ‘table assembly’ example to illustrate the algorithm, where the query was “What tool is needed to assemble a table?” (Answer: a hammer). The query is stated as an assertion (“there is an assembly of a table”), which forms the initial scenario CG, plus a path to the variable of interest (“what is the instrument of the assembly?”):

Initial Scenario CG:

Path to the variable of interest:



Given a CG knowledge-base which includes the CGs shown in Figure 5, the algorithm proceeds as follows:

1. For the first (and only) predicate in the path, $\text{inst}(\text{Assemble01}, y)$:

Classify:

First, the algorithm tries to refine the class of `Assemble01` from `ASSEMBLE` to something more specific. No refinements are found.

Elaborate:

As an `INST` arc from `Assemble01` is not yet contained in the scenario CG, it searches for generalizations of `Assemble01` whose schema contain information about the `INST` relation. One such schema is `ASSEMBLE(x)`, shown in Figure 5, and so it adds this arc to the scenario:



Now this arc can be traversed in the scenario CG. The algorithm thus finds that the object y at the end of this arc is:

$y = \text{Assemble01.ADHESIVE.APPLICATOR}$
 = “the thing which applies the adhesive in this assembly event”
 = the path $\text{adhesive}(\text{Assemble01}, x_1), \text{applicator}(x_1, x_2)$

So it recursively calls `evaluate_path((adhesive(Assemble01,x1), applicator(x1,x2)),CG)` to find the individual(s) this path refers to (ie. what the applicator is):

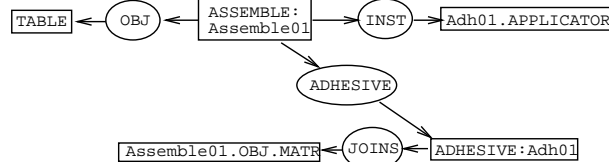
1.1 For the first predicate in this path, `adhesive(Assemble01,y)` (“What is the adhesive used in the assembly?”):

Classify:

First, the algorithm tries to refine the class of `Assemble01` from `ASSEMBLE` to something more specific. No refinements are found.

Elaborate:

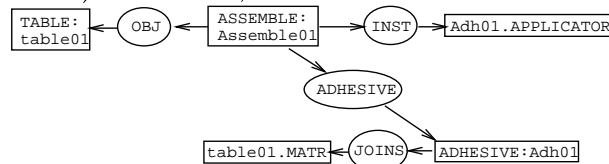
As the `ADHESIVE` arc is not yet contained in the scenario, the algorithm searches for generalizations of `Assemble01` whose schema contain information about the `ADHESIVE` relation. One such schema is again `ASSEMBLE(x)`. From this schema, `y` is found to be an instance of (the concept) `ADHESIVE` which `JOINS` the material of the assembled object. This part of the schema is joined with the scenario `CG`, and a Skolem individual `Adh01` is created to label this instance:



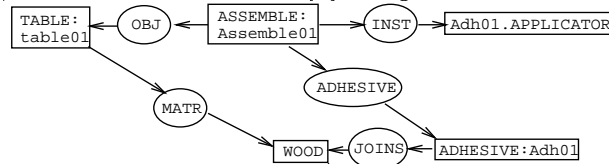
1.2 For the second predicate in the path, `applicator(Adh01,y)` (“What is used to apply this adhesive?”):

Classify:

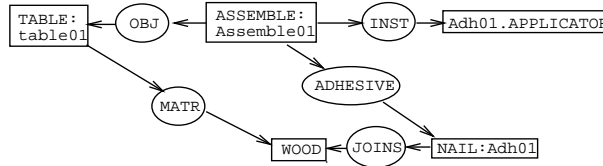
First, the algorithm tries to refine the class of `Adh01` from `ADHESIVE` to something more specific. By searching type definitions in the KB, the algorithm finds that one possible specialization of `ADHESIVE` is `NAIL`, defined as an `ADHESIVE` which joins `WOOD`. To test if this applies, it recursively calls the algorithm to find `Adh01.JOINS` (“What does the adhesive join?”). Traversing this arc in the scenario, the algorithm finds the path `Assemble01.OBJ.MATR`, and hence the algorithm is again called recursively to evaluate this path. First, `Assemble01.OBJ` is found (in the scenario `CG`) to be `TABLE`, hence a Skolem constant `Table01` is created:



Then, `Table01.MATR` is found by joining in the schema for `TABLE`:

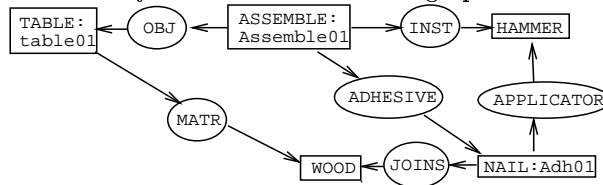


Hence the value `Wood01` is returned as the thing which `Adh01` joins. Hence the definition is satisfied, and `Adh01`'s type can be refined from `ADHESIVE` to `NAIL`.



Elaborate:

As the `APPLICATOR` arc is not yet contained in the scenario, the algorithm searches for generalizations of `Adh01` whose schema contain information about the `APPLICATOR` relation. One such schema is `NAIL(x)`, shown in Figure 5. This is joined with the scenario graph:



Traversing this arc, the node `HAMMER` (denoting an instance of hammer) is found. A Skolem individual `Hammer01` is again created to label that instance.

- 1.3 Hence return $y = \text{Hammer01}$ (an instance of `HAMMER`)
- 2. Hence return $y = \text{Hammer01}$ (an instance of `HAMMER`)

Note that the conclusion (that the `APPLICATOR` of `Adh01` is a `HAMMER`) results from the refinement of `Adh01`'s type from `ADHESIVE` to `NAIL`. Without the classify step in Figure 4, the inference engine would have failed to answer this query.

5 Discussion

5.1 Sources of Incompleteness

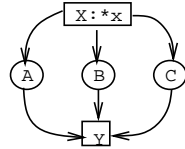
Our use of access paths is inspired by the work on Algernon [9], which similarly uses incomplete reasoning and access paths to achieve tractable inference. However, our work has evolved to exploit different properties of Algernon compared with those that Algernon was originally based on. In particular, the original research proved polynomial time inference for ‘pure’ Algernon, which did not include statements involving existential quantification or restrictions on the cardinality of relations (and thus is similar to Datalog [14]), and achieved tractability by limiting the number of bindings a variable in a rule could take [1]. In contrast, our interest has been in ‘full’ Algernon (due to our requirements for these features), and we exploit different sources of incompleteness, and hence efficiency, in its reasoning.

There are three sources of incompleteness in full Algernon which we exploit.

Consider the following schema with access paths:

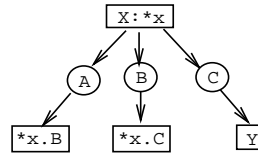
Standard CG Schema:

schema for $X(x)$ is



Schema with access paths:

schema for $X(x)$ is



Let $X0, Y0$ be instances of types X, Y respectively. Again assuming all these relations are all functional (Section 2.3), access paths limit inference in the following three ways:

Excluded Paths: The algorithm will try computing $C(X0, y)$ to find $B(X0, y)$, but not try $A(X0, y)$.

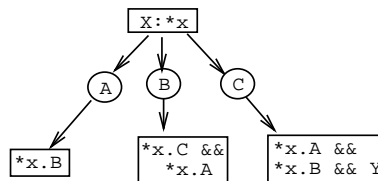
Directionality of Coreference: The algorithm will try computing $C(X0, y)$ to find $B(X0, y)$, but not the converse, ie. will not try computing $B(X0, y)$ to find $C(X0, y)$. (This can be viewed as a special case of excluded paths).

Hiding of Inverses: Given the inverse relation of B is B^{-1} , the algorithm will not infer from $C(X0, Y0)$ that $B^{-1}(Y0, X0)$ is also true³.

In all these cases, these inferences logically follow from the information in initial schema, plus the constraint that the relations are functional. However, the algorithm is (deliberately) not ‘clever’ enough to draw these conclusions (if it did, then access paths would be merely a notational convenience for indicating coreferential variables). In terms of the inference algorithm, the incompleteness arises due to the query path not passing through the node containing the access path, hence the access path is not evaluated, hence the information it encodes is not seen by the inference engine.

We are exploiting this property to allow the knowledge engineer to focus reasoning with CGs, so that inference follows just those access paths that he/she specifies. It is important to note the knowledge engineer can exploit this incompleteness as much or as little as desired, to hide parts of the knowledge base which would be undesirable to visit when answering a particular question. If such masking is not desired, suitable additional paths can always be added. Our implementation allows multiple alternative paths to be specified at a node (separated by the symbol $\&\&$), and similarly for paths and types to be mixed. For example, the above schema for X could be expressed with less incompleteness as:

schema for $X(x)$ is



³ eg. the algorithm will not be able to infer $[Car1] \rightarrow (DRIVEN-BY) \rightarrow [Fred]$ from $[PERSON: *p] \rightarrow (DRIVES) \rightarrow [*p.CAR]$ and $[Fred] \rightarrow (CAR) \rightarrow [Car1]$.

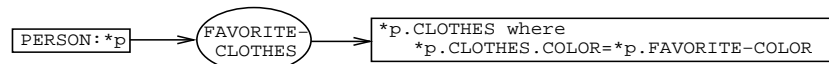
This provides additional paths by which the values of A , B and C can be found. With multiple paths at a node, the algorithm tries each in turn until a solution is found⁴. A looping-checker prevents infinite recursion.

Finally, it is important to note that, although inference may be incomplete when answering a particular question, there is no logical consequence of the knowledge base which is inherently unreachable: leading questions can always be asked which touch the required access paths, hence causing them to be evaluated and the information they contain to be added to the scenario. This property, in which all logical consequences are inferable by *some* sequence of questions, is referred to as “Socratic completeness” in access-limited logic [9].

5.2 Extensions

Access paths can be viewed as a special case of defining a CG node’s value as a computation. Based on this view, we can extend the path language, and hence the query language also, to include other forms of computation besides chains of relations. The full path language we use is described in [4], and includes additional constructs for equality, negation (as failure), conditionals, arithmetic, and filtering of values. We can then express, for example, that a person’s favorite clothes are those which are his/her favorite color by placing an expression in the node at the end of the FAVORITE-CLOTHES arc in the PERSON schema, eg.

schema for PERSON(p) is



6 Summary and Conclusions

The most important result of this work is an inference method which makes reasoning with CGs possible in reasonable run-time. On a DEC Alpha 3000/500, our implementation runs at approximately 600 inferences per cpu-second, where an inference consists of making a single step along a path (which itself may require joining schema/type information to the conceptual graph being queried). More important than absolute speed, however, is the fact that inference is channeled down specific chains of subgoals, as specified by the access paths. This allows queries requiring complex composition of CGs to be answered quickly.

The cost of tractability is the loss of guarantee that the inference algorithm will derive all conclusions implied by the knowledge-base (though of course a time-bounded general inference algorithm will similarly lose this guarantee, which is perhaps a fairer comparison to make). The degree to which incompleteness is a practical problem depends crucially on whether an adequate set of access paths can be encoded, such that logical incompleteness has minimal effect on question-answering ability. Our experience is that this is generally possible; however, we are currently lacking adequate theoretical and empirical frameworks to characterize this, in particular to characterize the questions which can be answered, and the degree of speed-up which is thus achieved. Unfortunately, there

⁴ This is an incompleteness in our implementation: strictly, it should try all paths and then unify the result.

are few theoretical frameworks available which distinguish between the pragmatically different cases we are interested in (namely, how the degree of incompleteness impacts coverage and efficiency of answering questions); many frameworks (eg. worst-case complexity analysis) collapse the distinction that we are trying to clarify. Similarly, empirical analysis is difficult as we lack good methods for characterizing the space of questions that we are interested in answering, and good methods for accounting for the impact of the knowledge engineer in such an investigation (eg. guarding against the knowledge engineer “fixing” the knowledge-base to answer just those questions). These are complex but important issues for further investigation.

Acknowledgements: Support for this research was provided by a grant from Digital Equipment Corporation and a contract from the Air Force Office of Scientific Research (F49620-93-1-0239). This work was conducted at the University of Texas at Austin.

References

1. James Crawford. Access-limited logic: A language for knowledge representation. Technical Report AI90-141, Dept CS, Univ Texas at Austin, Austin, TX, Oct 1990.
2. B. W. Porter, J. Lester, K. Murray, K. Pittman, A. Souther, L. Acker, and T. Jones. The botany knowledge base project. Tech Report AI-88-88, Dept CS, Univ Texas at Austin, Sept 1988.
3. Peter Clark and Bruce Porter. The dce help-desk project. (<http://www.cs.utexas.edu/users/mfkb/dce.html>), 1996.
4. Peter Clark. KM/KQL: Syntax and semantics. (Internal document, AI Lab, Univ Texas at Austin. <http://www.cs.utexas.edu/users/mfkb/manuals/kql.ps>), 1996.
5. J. F. Sowa. *Conceptual structures* Addison Wesley, 1984.
6. R. J. Brachman, D. L. McGuinness, P. F. Patel-Schneider, L. A. Resnick, and A. Borgida. Living with CLASSIC: When and how to use a KL-ONE like language. In J. Sowa, editor, *Principles of Semantic Networks*. Kaufmann, CA, 1991.
7. Adil Kabbaj. Prolog cg-object-oriented programming with PROLOG+CG. Master’s thesis, Univ Montreal, Canada, May 1995.
8. Jon Doyle and Ramesh S. Patil. Two theses of knowledge representation: Language restrictions, taxonomic classification, and the utility of representation services. *Artificial Intelligence*, pages 261–297, 1991.
9. J. M. Crawford and B. J. Kuipers. Algernon – a tractable system for knowledge-representation. *SIGART Bulletin*, 2(3):35–44, June 1991.
10. R. MacGregor. Loom users guide (version 1.4). Tech report, ISI, CA, 1991.
11. Ronald J. Brachman, Richard E. Fikes, and Hector J. Levesque. KRYPTON: A functional approach to knowledge representation. In Ronald J. Brachman and Hector J. Levesque, editors, *Readings in Knowledge Representation*, pages 412–429. Kaufmann, CA, 1985.
12. William A. Woods. Understanding subsumption and taxonomy. In John F. Sowa, editor, *Principles of Semantic Networks*, pages 45–94. Kaufmann, CA, 1991.
13. Hassan Ait-Kaci and Andreas Podelski. Towards a meaning of LIFE. *Logic Programming*, 16:195–234, 1993.
14. Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, Mar 1989.

This article was processed using the L^AT_EX macro package with LLNCS style