Representing Conceptual Graphs in Algernon: Working Note 9

Peter Clark and Ben Kuipers
Dept. Computer Science
Univ. Texas at Austin
Austin, TX 78712, USA

1 Introduction

"Conceptual Graphs" have become a popular representational formalism in AI research. They were introduced by Sowa in his 1984 book "Conceptual Structures" [Sowa, 1984], as a modern treatment of ideas from the 19th century philosopher Peirce.

Conceptual graphs are also the fundamental representational unit in KM/KQL [Clark, 1996], the language of the Botany KB project. In KM, these units are referred to as 'frames'; Each frame encodes a CG along with addition information, in the form of access paths, about how to infer values of particular nodes (see Section 3.2).

This working note describes now conceptual graphs can be mapped onto the representation language Algernon [Crawford and Kuipers, 1991, Crawford, 1990]. Section 3 describes one way in which CGs can be encoded in Algernon, where a set of rules encodes the various implications which a CG contains. Section 4 discusses this and alternative ways that CGs could be encoded with Algernon, and their relative merits and weaknesses.

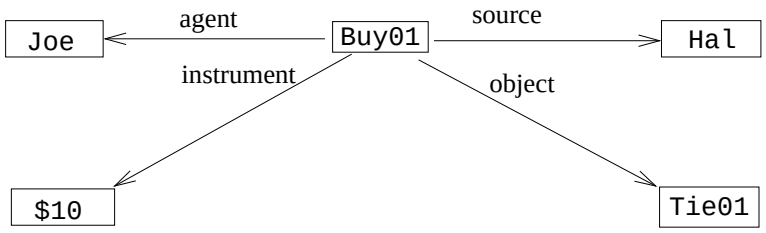
2 Introduction to CGs

Conceptual graphs are sometimes described as a "graphical notation for logic". While this is true (there are well-defined semantics), it doesn't really tell the whole story: CGs encourage a particular style of conceptualizing and representing knowledge.

There are two key concepts in CGs:
Instance Graphs: A network of instances and their relationships.
Type Definitions: Express properties of general concepts.

2.1 Instance Graphs

The basic conceptual graph can be thought of a set of instances and and their inter-relationships. The graph simply asserts that those instances and relationships exist. For example "Joe buys a necktie for $10" (from [Sowa, 1984] p110) might look:



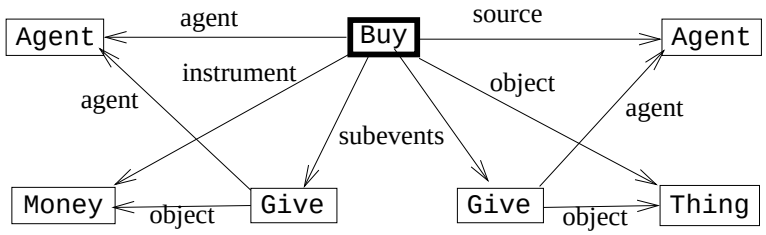
2.2 Type Definitions

A type definition describes the properties implied by an instance's class membership (ie. type). A type definition is essentially an inference rule that says:

```
IF   an instance of type T exists
THEN it will be in a specific set of relationships to other instances.
```

This latter "set of relationships" is itself expressed as a CG. In other words, parts of one CG may imply another CG also holds.

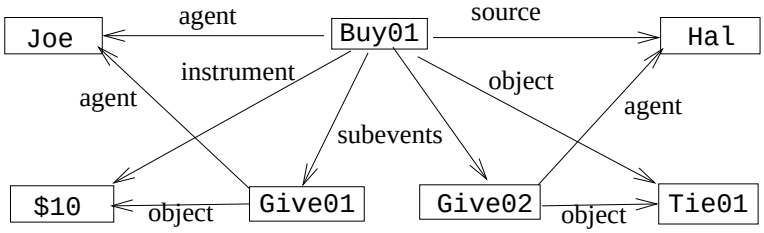
For example, the type definition for `Buy` includes (modified from [Sowa, 1984] p110):



stating that "IF there is a Buy, THEN there is a Give (which is the subevent of the Buy), and some Money (which is the instrument of the Buy), and the given object in that Give = the Money instrument in that Buy, and ...".

2.3 Joins: Inference with CGs

If a type definition is found to apply to some individual in the instance graph, then we can "merge" the implied CG with that initial graph. This merging process is often referred to as a **graph unification** or **maximal join** or just a **join**. By matching up relations in the initial instance graph with instances in the implied CG, we can determine that certain nodes are coreferential[1]. For example, joining the earlier type definition for `Buy` with the earlier instance graph, we get:



Thus (the implied instance of) Money (implied by `Buy`) and `$10` are considered coreferential (they are both the unique instrument of the Buy) and hence are merged.

[1] A 'maximal join' is a join where coreferentiality is always assumed, if there is some question about whether two nodes are coreferential or not.

Joins are the basic inference rule in CGs, and are the graphical equivalent of modus ponens. By applying joins, we can elaborate an instance graph to deduce additional facts and relationships about the individuals it contains.

3 Conceptual Graphs in Algernon and KM

A graph join is in fact equivalent to performing logical deduction, paying special attention to implied coreferentiality. A type definition can be thought of simply as a packet of inference rules, nicely encapsulated into a single syntactic structure. For example, the earlier type definition for `Buy` encodes the inference rules:

```
IF   (isa $X Buy)
THEN there exists some money $Money s.t. (instrument $X $Money).

IF (isa $X Buy)
THEN there exists a give $Give s.t. (subevent $X $Give).

IF (isa $X Buy) and (instrument $X $Money) and (subevent $X $Give)
THEN (object $Give $Money).

etc.
```

KM similarly encapsulates such rules into a single syntactic structure, namely a KM frame, as illustrated shortly.

This equivalence of CGs to a "rule packet" gives us a basis for expressing CGs in Algernon – we can simply encode the rules as Algernon rules. Although the rules are not encapsulated in a single syntactic structure, they are largely functionally equivalent (see Section 3.2 for caveats).

Doing a CG join corresponds to firing all the rules representing a CG, so as to elaborate an initial graph with facts implied by those rules. However, in practice it may not be necessary to fire all the rules to answer a query – instead, the system could fire just those rules pertinent to a query, and hence avoid unnecessary work. This in fact is what both KM's and Algernon's basic inference mechanism do, and so we can view them as doing *partial joins* or *partial unification*, allowing the goal query to drive the firing of rules. In this way, the instance graph is just elaborated enough to answer some query posed to the system, This turns out to work nicely.

3.1 An Example

The following illustrates how CGs can be represented in Algernon and KM. It is a modification of Sowa's "Buy" example (p110 [Sowa, 1984]). Figure 1 shows CGs for the concepts `Tie` (as in clothing), `Give`, `Get`, and `Buy`, and an instance graph describing the situation of "Joe buying a tie from Hal.". By *joining* the graphs, the initial instance graph is elaborated[2] to produce the graph shown lowest. That graph contains additional information about the situation, including deductions that:

- the instrument of the Buy is $10.
- Joe has $10[3].
- Hal and Joe are at the same place.

[2] In fact, specialized.

[3] at the start of the situation; this simple representation side-steps the complexities of representing change with time.

Given a query (eg. "How much does Joe have?"), joins such as those illustrated can be performed to allow the answer to be concluded. I'm currently not sure how, or even whether, a CG inference engine (eg. as in the Peirce system[Ellis, 1996]) determines which joins are required to answer a query. However it turns out that using access paths in Algernon and KM provide nice answers to this question: paths provide a sequence of sub-goals for the inference engine to solve in order to answer a top-level query, as described later in Section 3.2.

Figures 2, 3, 4, 5, and 6 illustrate how these various CGs can be represented in both Algernon and KM. KM retains the CG as a single syntactic structure whereas Algernon "busts it up" into a set of rules.

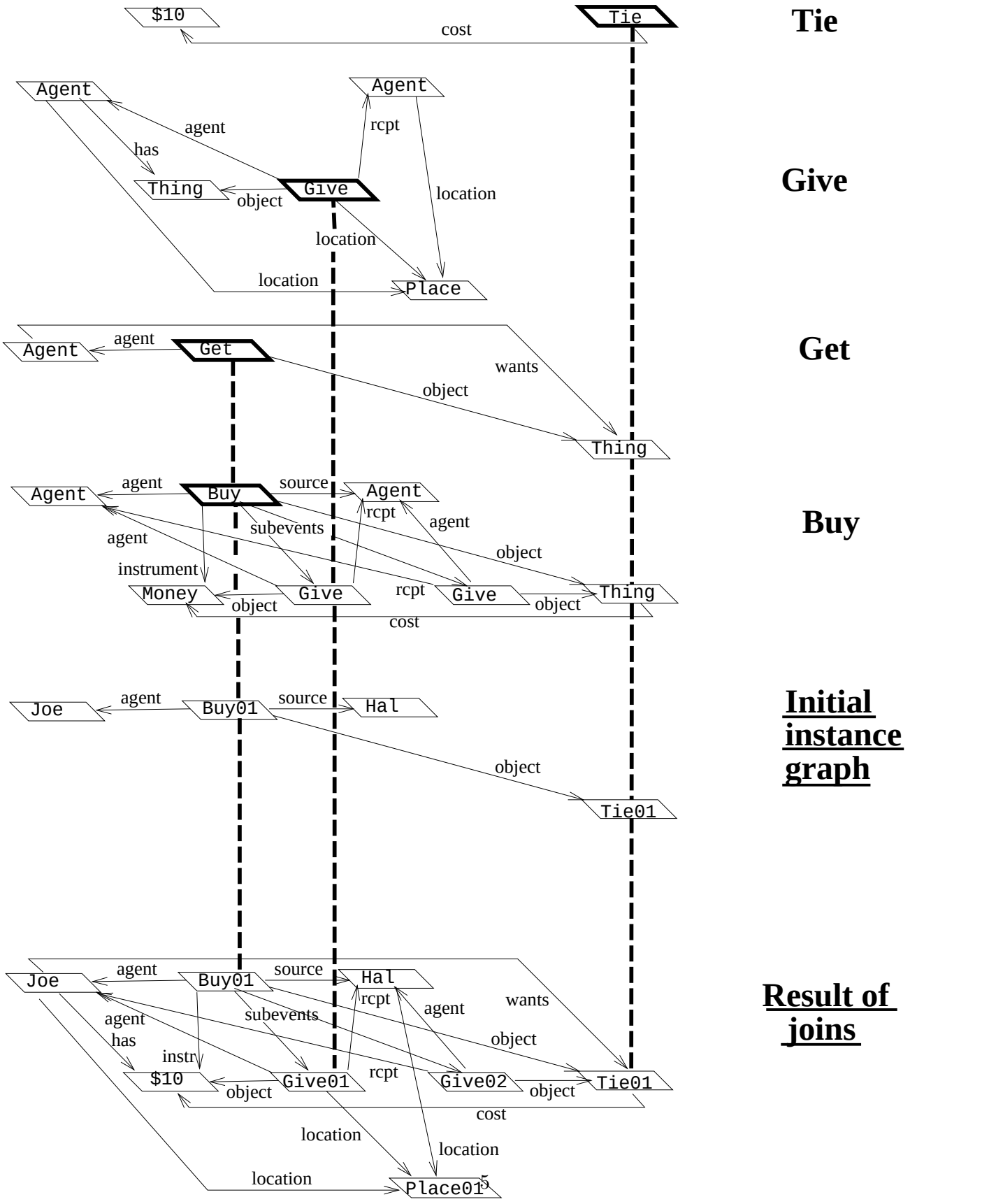


Figure 1: Five conceptual graphs and the result of their join.

(a) Conceptual Graph

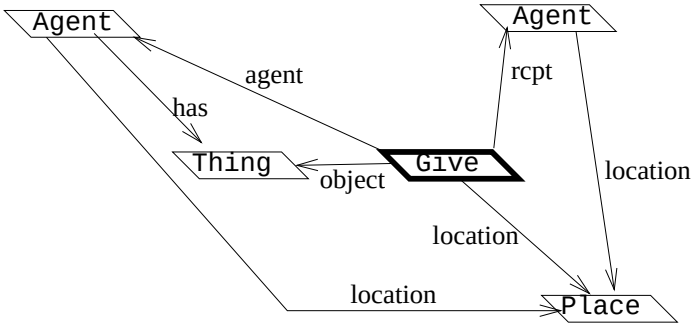
```
(:rules Ties ((cost ?t ?d) <- (:forc ?d (cost ?t ?d) (isa ?d TenDollars))))
```

(b) Algernon

```
(Tie
  (cost (TenDollars)))
```

(c) KM

Figure 2: Representations of `Necktie`.



(a) Conceptual Graph

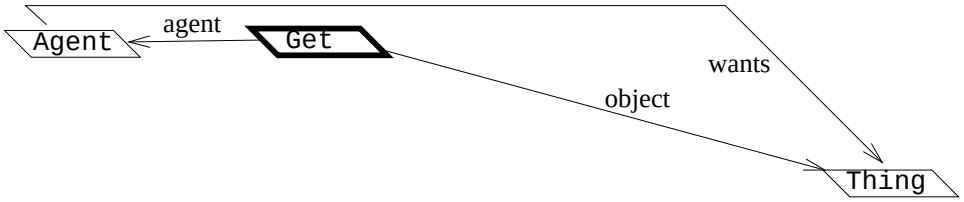
```
(:rules Gives ((agent ?b ?a) <- (:forc ?a (agent ?b ?a) (isa ?a Agents)))
              ((object ?b ?t) <- (:forc ?t (object ?b ?t) (isa ?t Things)))
              ((rcpt ?b ?s) <- (:forc ?s (rcpt ?b ?s) (isa ?s Agents)))
              ((location ?b ?m) <- (:forc ?m (location ?b ?m) (isa ?m Places))))
(:rules Agents ((has      ?a ?t) <- (agent-of ?a ?g) (isa ?g Gives) (object ?g ?t))
               ((location ?a ?t) <- (agent-of ?a ?g) (isa ?g Gives) (location ?g ?t))
               ((location ?a ?t) <- (rcpt-of  ?a ?g) (isa ?g Gives) (location ?g ?t)))
```

(b) Algernon

```
(Give
  (object (Thing))
  (location (Place))
  (agent ((Agent
            (has ((Self object)))            ; <- means "the object of Self"
            (location ((Self location))))))  ; where Self = the instance of
  (rcpt ((Agent                              ;              Give in question
            (location ((Self location)))))))
```

(c) KM

Figure 3: Representations of `Give`.



(a) Conceptual Graph

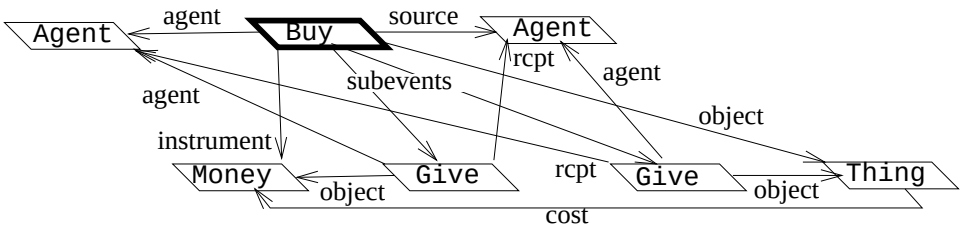
```
(:rules Gets ((agent  ?g ?a) <- (:forc ?a (agent ?g ?a) (isa ?a Agents)))
             ((object ?g ?t) <- (:forc ?t (object ?g ?t) (isa ?t Things))))
(:rules Agents ((wants ?a ?t) <- (agent-of ?a ?g) (isa ?g Gets) (object ?g ?t)))
```

(b) Algernon

```
(Get
  (object (Thing))
  (agent ((Agent
             (wants ((Self object Thing)))))))
```

(c) KM

Figure 4: Representations of `Get`.



(a) Conceptual Graph

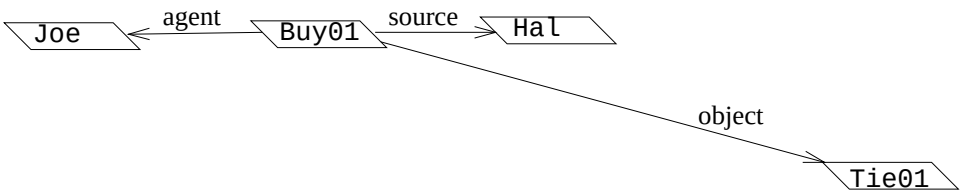
```
(:rules Buys ((agent ?b ?a) <- (:forc ?a (agent ?b ?a) (isa ?a Agents)))
             ((object ?b ?t) <- (:forc ?t (object ?b ?t) (isa ?t Things)))
             ((source ?b ?s) <- (:forc ?s (source ?b ?s) (isa ?s Agents)))
             ((instrument ?b ?m) <- (:or ((:retrieve (instrument ?b ?m)))       ; find it
                                         ((object ?b ?t) (cost ?t ?m))          ; compute it
                                         ((:a ?m (isa ?m Moneys)))))            ; create it
             ((subevent ?b ?g) <- (:forc ?g (subevent ?b ?g) (position ?g 1) (isa ?g Gives)))
             ((subevent ?b ?g) <- (:forc ?g (subevent ?b ?g) (position ?g 2) (isa ?g Gives))))
(:rules Gives ((agent  ?g ?a) <- (subevent-of ?g ?b) (isa ?b Buys) (position ?g 1) (agent ?b ?a))
              ((object ?g ?a) <- (subevent-of ?g ?b) (isa ?b Buys) (position ?g 1) (instrument ?b ?a))
              ((rcpt   ?g ?a) <- (subevent-of ?g ?b) (isa ?b Buys) (position ?g 1) (source ?b ?a))
              ((agent  ?g ?a) <- (subevent-of ?g ?b) (isa ?b Buys) (position ?g 2) (source ?b ?a))
              ((object ?g ?a) <- (subevent-of ?g ?b) (isa ?b Buys) (position ?g 2) (object ?b ?a))
               ((rcpt   ?g ?a) <- (subevent-of ?g ?b) (isa ?b Buys) (position ?g 2) (agent ?b ?a)))
```

(b) Algernon

```
(Buy
  (agent (Agent))
  (object (Thing))
  (source (Agent))
  (instrument ((Self object Thing cost Moneys)))
  (subevent ((Give
               (agent ((Self agent)))
               (object ((Self instrument)))
               (rcpt ((Self source))))
             (Give
               (agent ((Self source)))
               (object ((Self object)))
               (rcpt ((Self agent)))))))
```

(c) KM

Figure 5: Representations of Buy.



(a) Conceptual Graph

```
((:a ?j (isa ?j Agents) (name ?j "Joe"))
 (:a ?h (isa ?h Agents) (name ?h "Hal"))
 (:a ?t (isa ?t Ties)   (name ?t "a necktie"))
 (:a ?b (isa ?b Buys)   (name ?b "buy01"))
 (agent ?b ?j)
 (source ?b ?h)
 (object ?b ?t)))
```

(b) Algernon

```
 (Buy01               ; Buy01, Joe, Hal are flagged as instances in the KB
    (agent (Joe))
    (object (Tie))
    (source (Hal)))
```

(c) KM

Figure 6: Representations of `Buy01`, a buying situation.

3.2 Adding Access Paths to CGs

3.2.1 Guiding Inference

In fact, the Algernon and KM representations add an important element to CGs, namely *access paths*. Consider (part of) the representation of `Buy` (Figure 5) in Algernon:

```
(:rules Buys ((instrument ?b ?m) <- (object ?b ?t) (cost ?t ?m)))
```

and KM:

```
(Buy
  (object (Thing))
  (instrument ((Self object Thing cost Moneys)))
```

In both representations, the instrument (ie. money) of the Buy is specified as a path to the cost of the object (ie. purchase) of the Buy:

```
(object ?b ?t) (cost ?t ?m)             ; Algernon
(Self object Thing cost Moneys)         ; KM
```

where `Self` in KM is equivalent to the `?b` in Algernon.

Note that this path expresses more than the equivalence of the Buy's instrument with the Buy's object's cost; it also expresses a path – a series of subgoals – by which that instrument can be computed, namely:

1. Find the object of the Buy
2. Find the cost of that object

The advantage of this is that the path can drive inference: A query for "the instrument of Buy01" will proceed as follows:

1. From `Buys`, apply the rule
   `(:rules Buys ((instrument ?b ?m) <- (object ?b ?t) (cost ?t ?m)))`
2. `(object ?b ?t)` is already known (`(object buy01 tie01)`).
3. For `(cost tie01 ?m)`, apply the rule from `Ties`:
   `(:rules Ties ((cost ?t ?d) <- (:forc ?d (cost ?t ?d) (isa ?d TenDollars))))`
4. Hence the result `(instrument Buy01 TenDollars)` is found.

Note what has happened here: A rule fires (1), the path in its antecedent sets up subgoals, and (recursively) rules fire to solve those subgoals (3). This is equivalent to doing partial, demand-driven joins of conceptual graphs to elaborate the instance graph and find the answer to a query.

3.2.2 Performing Inference

The following illustrates doing inference using the CGs represented above. The definition of the `text` relation simply collects the names of objects together and concatenates them into a friendly string.

```
algy> q
 Path to query> ((:a ?b (isa ?b buys)) (instrument ?b ?m))
   Bindings:            ?m     --- frame4

algy> vf frame4
Frame4:
  Isa:        moneys physical-objects objects things
  Instrument-of: frame3
```

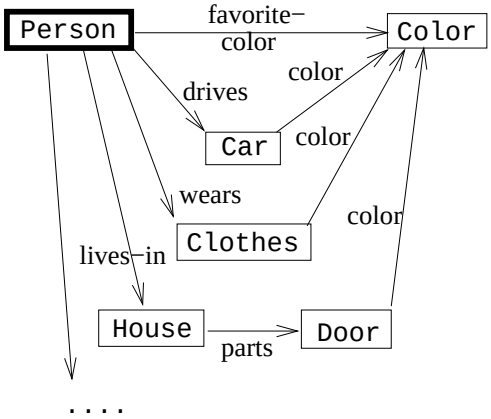
```
algy> q
 Path to query> ((:a ?t (isa ?t Ties))
                 (:a ?b (isa ?b Buys) (object ?b ?t))
                 (instrument ?b ?m))
   Bindings:            ?m     --- ten\ dollars1 "ten dollars"

algy> q
 Path to query> ((:a ?b (isa ?b Buys)) (text ?b ?t))
   Bindings:            ?t     --
"the AGENT has the MONEY. the AGENT has the PHYSICAL-OBJECT.
 the AGENT wants the PHYSICAL-OBJECT. the AGENT gives the MONEY to
 the AGENT. the AGENT gives the PHYSICAL-OBJECT to the AGENT."

algy> q
 Path to query> ((text buy01 ?t))
   Bindings:            ?t     --
"Joe has ten dollars. Hal has a necktie. Joe wants a necktie.
 Joe gives ten dollars to Hal. Hal gives a necktie to Joe."
```

3.2.3 Restricting Inference

Paths guide inference: if some node X is unknown, then a path can indicate which sequence of queries might find the target value. Consider the query `(favorite-color Pete ?c)`, given the CG:



There are several paths to – and hence ways of computing – the value of `?c`, including:

```
(drives Pete ?x) (color ?x ?c)
(wears Pete ?x) (color ?x ?c)
(lives-in Pete ?h) (parts ?h ?d) (isa ?d Doors) (color ?d ?c)
...
```

By giving a path, we guide the inference engine about which path to follow, for example:

```
(:rules People ((favorite-color ?p ?c) <- (wears ?p ?x) (color ?x ?c)))
```

This path is of course a restriction as well as a guide on inference. In this example, if the color of Pete's clothes is unknown then Algernon will fail to answer the question `(favorite-color Pete ?c)`, even if (say) the color of Pete's car known. I view this as an advantage: inference should be thought of as a resource-bounded process, and hence all inference process are in practice incomplete (due to time-bounds), even if theoretically complete. What matters, then, is "value per logical inference"; the hope is that access paths will result in a high such value.

Of course, Algernon and KM are not restricted to a single access path; we can enumerate as many as we like, eg. using multiple rules:

```
(:rules People ((favorite-color ?p ?c) <- (wears ?p ?x) (color ?x ?c))
               ((favorite-color ?p ?c) <- (drives ?p ?x) (color ?x ?c)))
```

or in KM:

```
(People
  (favorite-color ((Self drives Car color Color) &&
                   (Self wears Clothes color Color))))
```

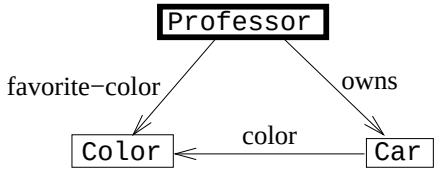
This of course makes the system 'more complete', but requiring more inference.

Another example of control with access paths is in Figure 3, where the Give's agent's location is specified as a path to the Give's location, but not as a path to the Give's recipient's location. Thus, if the system knew the latter (the Give's recipient's location), but not the former (the Give's location), then it would fail to compute the Give's agent's location, although declaratively the knowledge is there to compute an answer. As before, we can 'reduce the incompleteness' by adding in extra access paths, at the cost of making higher inference demands.

4 Alternative Ways of Representing CGs in Algernon

The discussion so far assumes that a CG is best represented in Algernon by breaking it up into a large number of rules. In fact, there are alternative methods: some of these are less verbose, and keep the rules for a CG nicely grouped together, but cause incompleteness of the inferencing. In fact, KM's internal implementation of CGs is most like that described in Section 4.3 below, and KM has the incompleteness described in that Section.

As a discussion example, consider representing the CG which expresses that "All professors own a car, whose color is their favorite color":



In KM this would be expressed:

```
(Professor
  (favorite-color (Color))
  (owns ((Car
           (color ((Self favorite-color)))))))
```

Again, `Self` is a reference to the instance of professor under consideration, and the expression `(Self favorite-color)` is an access path to that professor's favorite color (read this as "the favorite-color of Self").

4.1 Method 1: Bust Up into Rules

The method presented earlier took the approach of breaking up CGs into multiple rules. Here the three rules would be:

```
(:rules Professors ((owns ?p ?c) <- (:forc ?c (owns ?p ?c) (isa ?c Cars)))
                   ((favorite-color ?p ?o) <- (:forc ?c (favorite-color ?p ?o)
                                                        (isa ?o Colors))))
(:rules Cars ((color ?c ?o) <- (owned-by ?c ?p)
                               (isa ?p Professors)
                               (favorite-color ?p ?o)))
```

This approach has the advantage of being straight-forward and "inferentially complete", ie. various queries about professors' car color are all answered by this representation, including some which the inference procedures in KM fail to answer, as described later.

There are two significant disadvantages to this approach:

Rule Proliferation: General frames, such as `Cars`, are getting loaded with information about *specific types* of cars (eg. professors' cars), rather than just cars in general. This might be problematic for inference. Consider:

```
;;; "People give gifts made of gold at Western weddings"
(Give-At-Western-Wedding
  (agent (Person))
  (recipient (Person))
  (object ((Thing
             (made-of (Gold))))))
```

This would compile to rules in Algernon (under this method) including:

```
((:rules Things
         ((madeof ?t Gold) <- (object-of ?t ?g)
                              (isa ?g Gives-At-Western-Weddings))))
```

Now if we ask "what is a tree made of?"

```
(ask '((:a ?t (isa ?t Trees)) (made-of ?t ?m)) :collect '?m)
```

Algernon will ask (say):

- "Is the tree the object of a western wedding gift?"
- "Is the tree produced by a car factory?"
- "Is the tree the object of a throw in a baseball game?"
- ....

Algernon could ask a large number of rather obscure questions using this representational method given a large knowledge-base. This could seriously degrade the performance of Algernon, or even make it unusable.

Rule Dispersion: The knowledge has been dispersed in the KB. If we look at rules under `Professors`, we just find rule "professors own cars", while the rule about professors' car colors is now stored under `Cars`. This is a potential problem if we wanted Algernon to generate a description of professors: We don't just want to get the response "Professors own a car.". How could we we identify and reassemble all the 'professor' information again?

4.2 Method 2: Full Generation of Slot-Values

As an alternative, Algernon could generate the *full* description of a professor's car, including computing it's color, at the creation-time of that car instance. This contrasts with method 1, where the color of professors' cars' would only be computed on demand using a rule under `Cars`.

Using this method, the CG would be represented by a single rule:

```
IF    you ever want to know what a professor owns
THEN  create a car C with color = professor's favorite color.
```

which in Algernon would look:

```
(:rules Professors
   ((owns ?p ?c) <- (favorite-color ?p ?o)
                    (:forc ?c (owns ?p ?c) (isa ?c Cars) (color ?c ?o)))
   ((favorite-color ?p ?o) <- (:forc ?c (favorite-color ?p ?o)
                                        (isa ?o Colors))))
```

This approach overcomes the two main disadvantages of method 1 – there is no longer a proliferation of rules, and all the information has been localized under `Professors`. However, it has two disadvantages also:

1. This works fine if Algernon don't already know which car a particular professor owns – the rule will go and create one of the appropriate color.

   BUT: If Algernon already knows the car, eg. `Bruces_car`, but not its color, then Algernon will fail to conclude the color. For example, the query

   ```
   (color Bruces_car ?o)
   ```

   will fail, even though I know that Bruce is a professor and his favorite color is red. The reason is that the `(owns ...)` question is never asked (instead `(owns Bruce Bruces_car)` is asserted), and hence the find-or-create expression is never triggered.

2. Algernon will perform potentially redundant work, eg. the possibly complex computation of `(favorite-color ?p ?o)` is always done, even if color is not required to answer a question. (In this case, of course, that computation is trivial, but in general it could be arbitrarily complex).

4.3 Method 3: Same, but attach Rules to the generated Instances

Method 2 does potentially redundant work when generating the Skolem individuals (eg. generating an instance of `Cars` for a new professor); it may compute properties of those instances which are not needed to answer a particular question. We can avoid this by a variant, where instead of computing properties of new instances (eg. a car's color), we make Algernon attach *rules* to the new instance for computing these properties only on demand. The relation `selfset` is used to attach a rule to (the singleton set containing) an instance. It is a slightly unusual use of Algernon, involving a rule with two clauses in the consequent, one of which is the rule to assert.

```
(:rules Professors
   ((owns ?p ?c)
    (:rules ?c-set ((color ?c ?o) <- (favorite-color ?p ?o)))
    <-
    (:a ?c (isa ?c Cars))
    (selfset ?c ?c-set))
   ((favorite-color ?p ?o) <- (:forc ?c (favorite-color ?p ?o)
                                        (isa ?o Colors))))
```

In fact, this method is equivalent to the one used by KM/KQL. It avoids the proliferation of rules, but is syntactically still rather clunky. In addition, it still is inferentially incomplete in the same way as Method 2. KM/KQL similarly suffers from the same type of incompleteness.

4.4 Method 4: Forward-Chaining Rules

Methods 2 and 3 use a single rule to generate the full description (value plus all its idiosyncratic properties) of an instance should it be requested. As described in Section 4.2, this works fine if the instance doesn't already exist (the rule will generate one) but leads to inferential completeness if the instance *does* exist (the `bruces_car` example) but not all it's properties are known. In this latter case, the generative rule won't fire, and hence the properties of the instance won't get computed.

A final alternative would be to use forward-chaining rules to overcome this: As soon as an instance is created, *forward*-chain to compute all its associated properties; if any can't be computed then Algernon will set up continuations to ensure they are computed later when the required information becomes available. An encoding of the CG in this form would look:

```
(:rules Professors
        ((owns ?p ?c) <- (:forc ?c (owns ?p ?c) (isa ?c Cars)))
        ((owns ?p ?c) (favorite-color ?p ?o) -> (color ?c ?o))
        ((favorite-color ?p ?o) <- (:forc ?c (favorite-color ?p ?o)
                                             (isa ?o Colors))))
```

Now, if I assert `(owns Bruce Bruces_car)`, and Algernon can compute Bruce's favorite color, then this will work fine in this example.

There are still two disadvantages, however, with this solution:

1. The general form of this solution is to forward-chain to conclude all properties of all Skolem instances – a potentially intractible approach, and involving potentially unnecessary work. For example, the possibly expensive `favorite-color` computation will always be done, even that information is not required to answer a question.

2. There is a subtle incompleteness which can arise due to the interaction of forward- and backward- rules. This is described in more detail in the `order-dependence.lisp` example, and means that this representational method still can be inferentally incomplete.

4.5 Personal Reflection

From the point of view of inference, Method 1 has some nice features – in particular it overcomes some of the incompleteness of KM/KQL's inferencing (see example file `cg3.lisp`). However, it has the disadvantage of compiling a CG into a large number of rules.

From the syntactic point of view, this "clunkiness" could be easily overcome either by using some high-level language which compiled into Algernon, or using some representational constructs within Algernon to reduce the syntactic burden. For example, much of the syntactic verbosity could be removed using a rule of the form:

```
(:rules Instances
  ((?P ?x ?y) <- (type ?x ?type)
                 (defining-property ?type ?P ?restriction)
                 (:forc ?y (?P ?x ?y) (isa ?y ?restriction))))
```

Then types could be defined with properties and restrictions in a syntactically compact way that would set up the type and defining-property slots, for example:

```
(:deftype Buy
  (instrument Money)
  ...)
```

Then instances would be declared simply be asserting (isa ?x Instances). Although this doesn't handle internal constraints, it reduces much of the syntactic burden.

From the inferential point of view, a large number of rules may create scalability problems, as discussed in Section 4.1. This issue merits further investigation.

5 Conclusion

We have presented one way in which conceptual graphs can be encoded in Algernon, and then discussed this and three other alternative methods for encoding such graphs. This paper illustrates that conceptual graph structures can be handled in a rule-based framework, and also highlights some of the issues and trade-offs which occur when making such a translation.

References

[Clark, 1996] Clark, P. (1996). KM/KQL: Syntax and semantics. (Internal document, AI Lab, Univ Texas at Austin. http://www.cs.utexas.edu/users/mfkb/manuals/kql.ps).

[Crawford, 1990] Crawford, J. (1990). Access-limited logic: A language for knowledge representation. Technical Report AI90-141, Dept CS, Univ Texas at Austin, Austin, TX. (Phd thesis).

[Crawford and Kuipers, 1991] Crawford, J. M. and Kuipers, B. J. (1991). Algernon – a tractable system for knowledge-representation. *SIGART Bulletin*, 2(3):35–44.

[Ellis, 1996] Ellis, G. (1996). Peirce: A conceptual graphs workbench. http://www.cs.adelaide.edu.au:80/p̃eirce/.

[Sowa, 1984] Sowa, J. F. (1984). *Conceptual structures: Information processing in mind and machine*. Addison Wesley.