Conceptual Graphs

John F. Sowa

Abstract. A conceptual graph (CG) is a graph representation for logic based on the semantic networks of artificial intelligence and the existential graphs of Charles Sanders Peirce. Several versions of CGs have been designed and implemented over the past thirty years. The simplest are the typeless *core* CGs, which correspond to Peirce's original existential graphs. More common are the *extended* CGs, which are a typed superset of the core. The *research* CGs have explored novel techniques for reasoning, knowledge representation, and natural language semantics. The semantics of the core and extended CGs is defined by a formal mapping to and from ISO standard 24707 for Common Logic, but the research CGs are defined by a variety of formal and informal extensions. This article surveys the notation, applications, and reasoning methods used with CGs and their mapping to and from other versions of logic.

This is a preprint of Chapter 5 of the *Handbook of Knowledge Representation*, ed. by F. van Harmelen, V. Lifschitz, and B. Porter, Elsevier, 2008, pp. 213–237. It has been updated with some recent references and an Appendix with the CGIF grammar.

1. From Existential Graphs to Conceptual Graphs

During the 1960s, graph-based semantic representations were popular in both theoretical and computational linguistics. At one of the most impressive conferences of the decade, Margaret Masterman (1961) introduced a graph-based notation, called a *semantic network*, which included a lattice of concept types; Silvio Ceccato presented *correlational nets*, which were based on 56 different relations, including subtype, instance, part-whole, case relations, kinship relations, and various kinds of attributes; and David Hays presented *dependency graphs*, which formalized the notation developed by the linguist Lucien Tesnière (1959). The early graph notations represented the relational structures underlying natural language semantics, but none of them could express full first-order logic. Woods (1975) and McDermott (1976) wrote scathing critiques of their logical weaknesses.

In the late 1970s, many graph notations were designed to represent first-order logic or a formally-defined subset (Findler 1979). Sowa (1976) developed a version of *conceptual graphs* (CGs) as an intermediate language for mapping natural language questions and assertions to a relational database. Figure 1 shows a CG for the sentence *John is going to Boston by bus*. The rectangles are called *concepts*, and the circles are called *conceptual relations*. An arc pointing toward a circle marks the first *argument* of the relation, and an arc pointing away from a circle marks the last argument. If a relation has only one argument, the arrowhead is omitted. If a relation has more than two arguments, the arrowheads are replaced by integers 1,...,*n*.

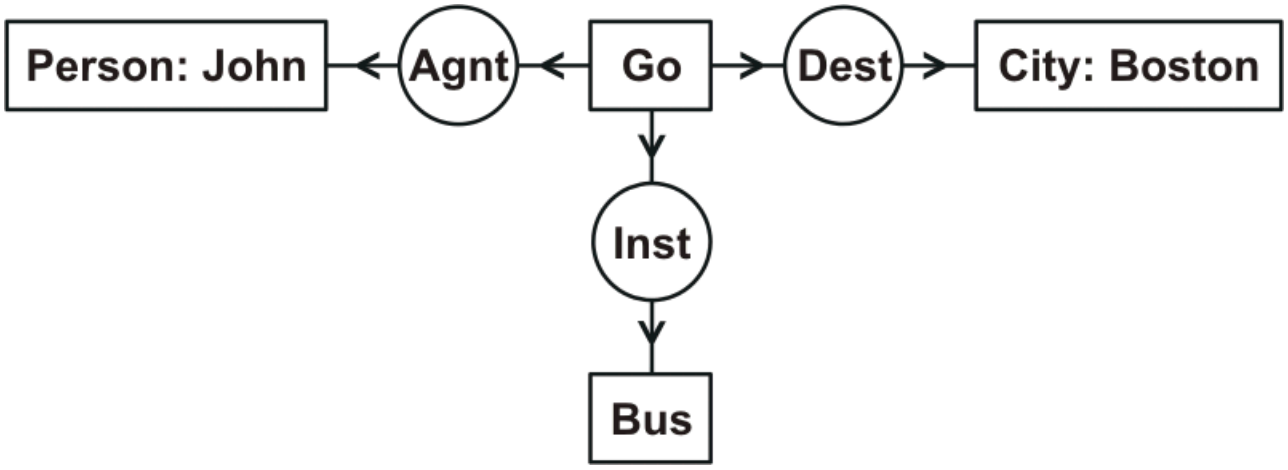

Figure 1: CG display form for *John is going to Boston by bus.*

The conceptual graph in Figure 1 represents a typed or sorted version of logic. Each of the four concepts has a *type label*, which represents the type of entity the concept refers to: `Person`, `Go`, `Boston`, or `Bus`. Two of the concepts have *names*, which identify the referent: `John` or `Boston`. Each of the three conceptual relations has a type label that represents the type of relation: agent (`Agnt`), destination (`Dest`), or instrument (`Inst`). The CG as a whole indicates that the person John is the agent of some instance of going, the city Boston is the destination, and a bus is the instrument. Figure 1 can be translated to the following formula:

$$(\exists x)(\exists y)(\text{Go}(x) \wedge \text{Person}(\text{John}) \wedge \text{City}(\text{Boston}) \wedge \text{Bus}(y) \wedge \text{Agnt}(x,\text{John}) \wedge \text{Dest}(x,\text{Boston}) \wedge \text{Inst}(x,y))$$

As this translation shows, the only logical operators used in Figure 1 are conjunction and the existential quantifier. Those two operators are the most common in translations from natural languages, and many of the early semantic networks could not represent any others.

For his pioneering *Begriffsschrift* (concept writing), Frege (1979) adopted a tree notation for representing full first-order logic, using only four operators: assertion (the "turnstile" operator ⊢), negation (a short vertical line), implication (a hook), and the universal quantifier (a cup containing the bound variable). Figure 2 shows the Begriffsschrift equivalent of Figure 1, and following is its translation to predicate calculus:

$$\sim(\forall x)(\forall y)(\text{Go}(x) \supset (\text{Person}(\text{John}) \supset (\text{City}(\text{Boston}) \supset (\text{Bus}(y) \supset (\text{Agnt}(x,\text{John}) \supset (\text{Dest}(x,\text{Boston}) \supset \sim\text{Inst}(x,y)))))))$$

Frege's choice of operators simplified his rules of inference, but they lead to awkward paraphrases: *It is false that for every x and y, if x is an instance of going then if John is a person then if Boston is a city then if y is a bus then if the agent of x is John then if the destination of x is Boston then the instrument of x is not y.*

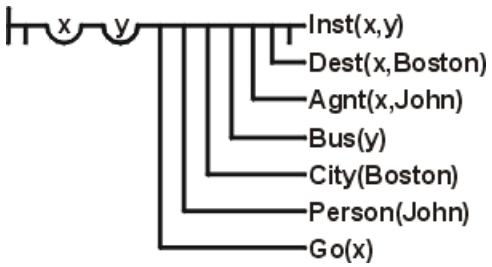

Figure 2: Frege's Begriffsschrift for *John is going to Boston by bus.*

Unlike Frege, who rejected Boolean algebra, Peirce developed the algebraic notation for first-order logic as a generalization of the Boolean operators. Since Boole treated disjunction as logical addition and conjunction as logical multiplication, Peirce (1880) represented the existential quantifier by Σ for repeated disjunction and the universal quantifier by Π for repeated conjunction. In the notation of Peirce (1885), Figure 1 could be represented

$$\Sigma_x\Sigma_y\,(\mathrm{Go}(x) \bullet \mathrm{Person}(\mathrm{John}) \bullet \mathrm{City}(\mathrm{Boston}) \bullet \mathrm{Bus}(y) \bullet \mathrm{Agnt}(x,\mathrm{John}) \bullet \mathrm{Dest}(x,\mathrm{Boston}) \bullet \mathrm{Inst}(x,y))$$

Peano adopted Peirce's notation, but he invented new symbols because he wanted to mix mathematical and logical symbols in the same formulas. Meanwhile, Peirce began to experiment with *relational graphs* for representing logic, as in Figure 3. In that graph, an existential quantifier is represented by *a line of identity*, and conjunction is the default Boolean operator. Since Peirce's graphs did not distinguish proper names, the monadic predicates isJohn and isBoston may be used to represent names. Following is the algebraic notation for Figure 3:

$$\Sigma_x\Sigma_y\Sigma_z\Sigma_w\,(\mathrm{Go}(x) \bullet \mathrm{Person}(y) \bullet \mathrm{isJohn}(y) \bullet \mathrm{City}(z) \bullet \mathrm{isBoston}(z) \bullet \mathrm{Bus}(w) \bullet \mathrm{Agnt}(x,y) \bullet \mathrm{Dest}(x,z) \bullet \mathrm{Inst}(x,w))$$

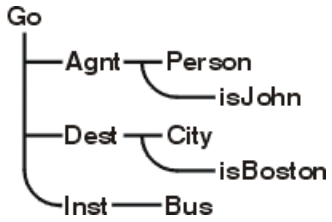

Figure 3: Peirce's relational graph for *John is going to Boston by bus.*

Peirce experimented with various graphic methods for representing the other operators of his algebraic notation, but like the AI researchers of the 1960s, he couldn't find a good way of expressing the scope of quantifiers and negation. In 1897, however, he discovered a simple, but brilliant innovation for his new version of *existential graphs* (EGs): an oval enclosure for showing scope. The default operator for an oval with no other marking is negation, but any metalevel relation can be linked to the oval. Sowa (1984) adopted Peirce's convention for CGs, but with rectangles instead of ovals: rectangles nest better than ovals; and more importantly, each context box can be interpreted as a concept box that contains a nested CG. A nest of two negations indicates a implication, as in Figure 4, which shows an EG and a CG for the sentence *If a farmer owns a donkey, then he beats it*.

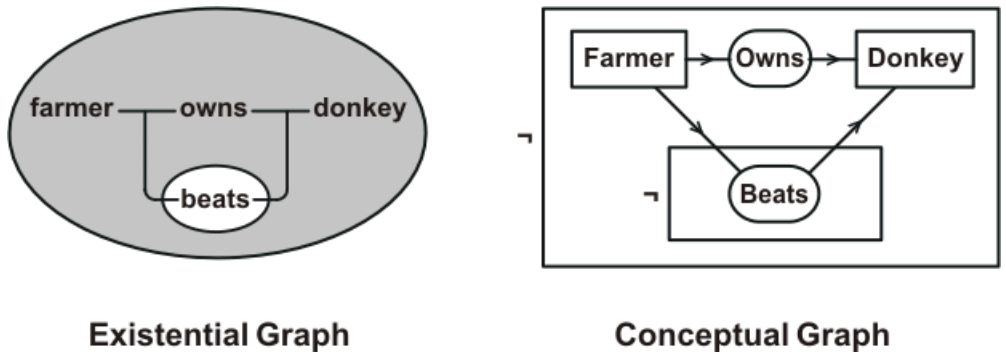

Figure 4: EG and CG for *If a farmer owns a donkey, then he beats it.*

To enhance the contrast between negative areas (nested in an odd number of ovals) and positive areas (nested in an even number of ovals), Peirce would shade the negative areas. As Figure 4 illustrates, the primary difference between EGs and CGs is the interpretation of the links: in EGs, each line of identity represents an existentially quantified variable, which is attached to the relations; in CGs, the concept boxes represent existential quantifiers, and the arcs merely link relation nodes to their arguments. Another difference is that the CG type labels become monadic relations in EGs. Unlike EGs, in which an unmarked oval represents negation, the symbol ~ marks a negated CG context. Both the EG and the CG could be represented by the following formula:

$$\sim(\exists x)(\exists y)(\text{Farmer}(x) \wedge \text{Donkey}(y) \wedge \text{Owns}(x,y) \wedge \sim\text{Beats}(x,y))$$

In order to preserve the correct scope of quantifiers, the implication operator $\supset$ cannot be used to represent the English *if-then* construction unless the existential quantifiers are moved to the front and converted to universals:

$$(\forall x)(\forall y)((\text{Farmer}(x) \wedge \text{Donkey}(y) \wedge \text{Owns}(x,y)) \supset \text{Beats}(x,y))$$

In English, this formula could be read *For every x and y, if x is a farmer who owns a donkey y, then x beats y.* The unusual nature of this paraphrase led Kamp (1981) to develop *discourse representation structures* (DRSs) whose logical structure is isomorphic to Peirce's existential graphs (Figure 5).

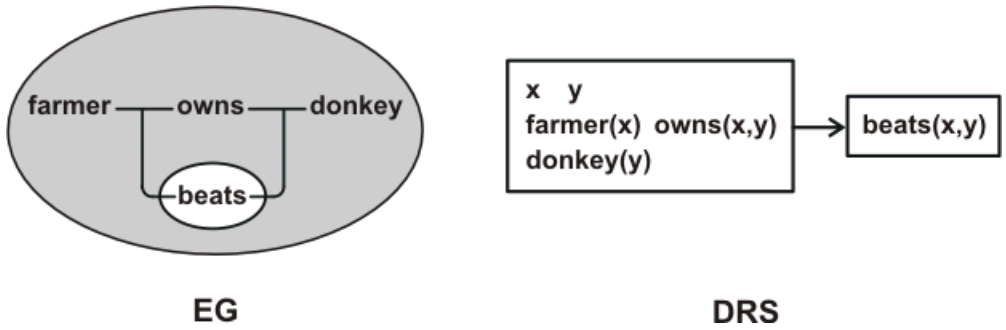

Figure 5: EG and DRS for *If a farmer owns a donkey, then he beats it.*

Kamp's primitives are the same as Peirce's: the default quantifier is the existential, and the default Boolean operator is conjunction; negation is represented by a context box, and implication is represented by two contexts. As Figure 5 illustrates, the nesting of Peirce's contexts allows the quantifiers in the antecedent of an implication to include the consequent within their scope. Although Kamp connected his boxes with an arrow, he made exactly the same assumption about the scope of quantifiers. Kamp and Reyle (1993) went much further than Peirce in analyzing discourse and

formulating the rules for interpreting anaphoric references, but any rule stated in terms of the DRS notation can also be applied to the EG or CG notation.

The CG in Figure 4 represents the verbs *owns* and *beats* as dyadic relations. That was the choice of relations selected by Kamp, and it can also be used with the EG or CG notation. Peirce, however, noted that the event or state expressed by a verb is also an entity that could be referenced by a quantified variable. That point was independently rediscovered by linguists, computational linguists, and philosophers such as Davidson (1967). The CG in Figure 6 shows a representation that treats events and states as entities linked to their participants by *case relations* or *thematic roles*.

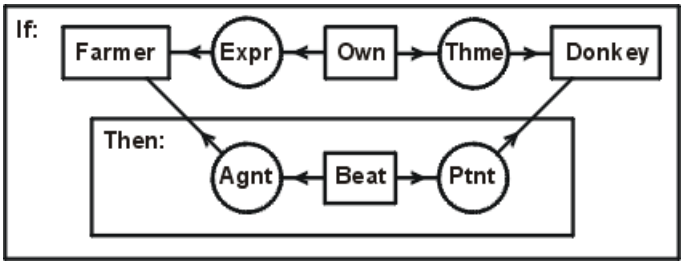

Figure 6: CG with case relations shown explicitly

The type labels `If` and `Then` in Figure 6 are defined as synonyms for negated contexts. The state of owning is linked to its participants by the relations experiencer (`Expr`) and theme (`Thme`), and the act of beating by the relations agent (`Agnt`) and patient (`Ptnt`). Following is the equivalent in typed predicate calculus:

$$\sim(\exists x{:}\text{Farmer})(\exists y{:}\text{Own})(\exists z{:}\text{Donkey})(\text{Expr}(y,x) \wedge (\text{Thme}(y,z) \wedge \sim(\exists w{:}\text{Beat})(\text{Agnt}(w,x) \wedge \text{Ptnt}(w,z)))$$

The model-theoretic semantics for the EGs and CGs shown in this section is specified in the ISO standard for Common Logic (CL). Section 2 of this article briefly describes the CL project, the CL model theory, and the mapping of the CL abstract syntax to and from the Conceptual Graph Interchange Format (CGIF), a linear notation that represents every semantic feature of the graphs. Section 3 presents the *canonical formation rules* for CGs and their use with Peirce's rules of inference for full FOL. Section 4 presents the use of CGs for representing propositions, situations, and metalevel reasoning. Section 5 discusses research issues that have inspired a variety of formal and informal extensions to the conceptual graph theory and notation.

2. Common Logic

Common Logic (CL) evolved from two projects to develop parallel ANSI standards for conceptual graphs and the Knowledge Interchange Format (Genesereth & Fikes 1992). Eventually, those projects were merged into a single ISO project to develop a common abstract syntax and model-theoretic foundation for a family of logic-based notations (ISO/IEC 24707). Hayes and Menzel (2001) defined a very general model theory for CL, which Hayes and McBride (2003) used to define the semantics for the languages RDF(S) and OWL. In addition to the abstract syntax and model theory, the CL standard specifies three concrete dialects that are capable of expressing the full CL semantics: the Common Logic Interchange Format (CLIF), the Conceptual Graph Interchange Format (CGIF), and the XML-based notation for CL (XCL). RDF and OWL can also be considered dialects that express subsets of the CL semantics: any statement in RDF or OWL can be translated to CLIF, CGIF, or XCL, but only a subset can be translated back to RDF or OWL.

The CL syntax allows quantifiers to range over functions and relations, but CL retains a first-order style of model theory and proof theory. To support a higher-order syntax, but without the computational complexity of higher-order semantics, the CL model theory uses a single domain D that includes individuals, functions, and relations. The option of limiting the domain of quantification to a single set was suggested by Quine (1954) and used in various theorem provers that allow quantifiers to range over relations (Chen et al., 1993).

Conceptual graphs had been a typed version of logic since the first publication in 1976, but Peirce's untyped existential graphs are sufficiently general to express the full CL semantics. Therefore, two versions of the Conceptual Graph Interchange Format are defined in the standard:

- **Core CGIF**. A typeless version of logic that expresses the full CL semantics. This dialect corresponds to Peirce's existential graphs: its only primitives are conjunction, negation, and the existential quantifier. It does permit quantifiers to range over relations, but Peirce also experimented with that option for EGs.
- **Extended CGIF**. An upward compatible extension of the core, which adds a universal quantifier; type labels for restricting the range of quantifiers; Boolean contexts with type labels `If`, `Then`, `Either`, `Or`, `Equivalence`, and `Iff`; and the option of importing external text into any CGIF text.

Although extended CGIF is a typed language, it is not *strongly typed*, because type labels are used only to restrict the range of quantifiers. Instead of causing a syntax error, as in the strongly typed logic Z (ISO/IEC 13568), a type mismatch in CGIF just causes the subexpression in which the mismatch occurs to be false. If a typed sentence in Z is translated to CGIF, it will have the same truth value in both languages, but a type mismatch, such as the following, is handled differently in each:

```
~[ [Elephant: 23] ]
```

This CGIF sentence, which is syntactically correct and semantically true, says that 23 is not an elephant. If translated to Z, however, the type mismatch would cause a syntax error. The more lenient method of handling types is necessary for representing sentences derived from other languages, both natural and artificial. RDF and OWL, for example, can be translated to CGIF and CLIF, but not to Z.

The conceptual graph in Figure 1, which represents the sentence *John is going to Boston by bus*, can be written in the following form in extended CGIF:

```
[Go *x] [Person: John] [City: Boston] [Bus *y]
(Agnt ?x John) (Dest ?x Boston) (Inst ?x ?y)
```

In CGIF, concepts are marked by square brackets, and conceptual relations are marked by parentheses. A character string prefixed with an asterisk, such as `*x`, marks a *defining node*, which may be referenced by the same string prefixed with a question mark, `?x`. These strings, which are called *name sequences* in Common Logic, represent *coreference labels* in CGIF and variables in other versions of logic. Following is the equivalent in CLIF:

```
(exists ((x Go) (y Bus))
   (and (Person John) (city Boston)
        (Agnt x John) (Dest x Boston) (Inst x y) ))
```

In the CL standard, extended CGIF is defined by a translation to core CGIF, which is defined by a translation to the CL abstract syntax. Following is the untyped core CGIF and the corresponding CLIF for the above examples:

```
[*x] [*y]
(Go ?x) (Person John) (City Boston) (Bus ?y)
(Agnt ?x John) (Dest ?x Boston) (Inst ?x ?y)

(exists (x y)
   (and (Go x) (Person John) (city Boston) (Bus y)
        (Agnt x John) (Dest x Boston) (Inst x y) ))
```

In core CGIF, the most common use for concept nodes is to represent existential quantifiers. A node such as [*x] corresponds to an EG line of identity, such as the one attached to the relation `Go` in Figure 3. It is permissible to write names in a concept node such as `[: John]`, but in most cases, such nodes are unnecessary because names can also be written in relation nodes. A concept node may contain more than one name or coreference label, such as `[: John ?z]`. In EGs, that node corresponds to a *ligature* that links two lines of identity; in CLIF, it corresponds to an equality: `(= John z)`.

Although CGIF and CLIF look similar, there are several fundamental differences:

1. Since CGIF is a serialized representation of a graph, labels such as `x` or `y` represent connections between nodes in CGIF, but variables in CLIF or predicate calculus.
2. Since the nodes of a graph have no inherent ordering, a CGIF sentence is an unordered list of nodes. Unless grouped by context brackets, the list may be permuted without affecting the semantics.
3. The CLIF operator `and` does not occur in CGIF because the conjunction of nodes within any context is implicit. Omitting the conjunction operator in CGIF tends to reduce the number of parentheses.
4. Since CGIF labels show connections of nodes, they may be omitted when they are not needed. One way to reduce the number of labels is to move concept nodes inside the parentheses of relation nodes:

   ```
   [Go *x]
      (Agnt ?x [Person: John])
      (Dest ?x [City: Boston])
      (Inst ?x [Bus])
   ```

   When written in this way, CGIF looks like a frame notation. It is, however, much more general than frames, since it can represent the full semantics of CL.

As another example, Figure 7 shows a CG for the sentence *If a cat is on a mat, then it is a happy pet.* The dotted line that connects the concept `[Cat]` to the concept `[Pet]`, which is called a *coreference link*, indicates that they both refer to the same entity. The `Attr` relation indicates that the cat, also called a pet, has an attribute, which is an instance of happiness.

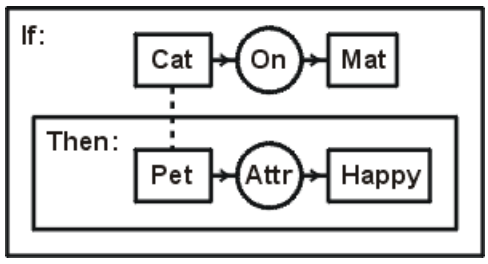

Figure 7: CG display form for *If a cat is on a mat, then it is a happy pet.*

The coreference link in Figure 7 is shown in CGIF by the defining label `*x` in the concept `[Cat: *x]` and the bound label `?x` in the concept `[Pet: ?x]`. Following is the extended CGIF and its translation to core CGIF:

```
[If: [Cat *x] [Mat *y] (On ?x ?y)
   [Then: [Pet ?x] [Happy *z] (Attr ?x ?z) ]]

~[ [*x] [*y] (Cat ?x) (Mat ?y) (On ?x ?y)
   ~[ (Pet ?x) [*z] (Happy ?z) (Attr ?x ?z) ]]
```

In CGs, functions are represented by conceptual relations called *actors*. Figure 8 is the CG display form for the following equation written in ordinary algebraic notation:

```
y = (x + 7)/sqrt(7)
```

The three functions in this equation would be represented by three actors, which are shown in Figure 8 as diamond-shaped nodes with the type labels `Add`, `Sqrt`, and `Divide`. The concept nodes contain the input and output values of the actors. The two empty concept nodes contain the output values of `Add` and `Sqrt`.

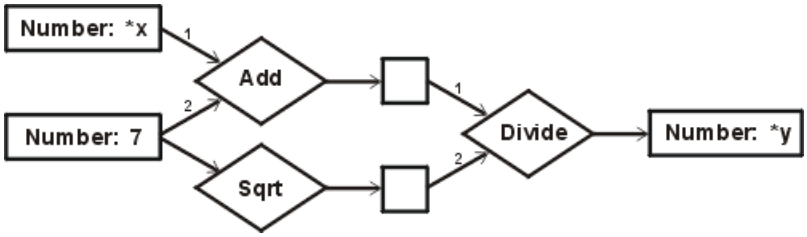

Figure 8: CL functions represented by actor nodes

In CGIF, actors are represented as relations with two kinds of arcs: a sequence of *input arcs* and a sequence of *output arcs*, which are separated by a vertical bar:

```
[Number: *x] [Number: *y] [Number: 7]
(Add ?x 7 | [*u]) (Sqrt 7 | [*v]) (Divide ?u ?v | ?y)
```

In the display form, the input arcs of `Add` and `Divide` are numbered 1 and 2 to indicate the order in which the arcs are written in CGIF. Following is the corresponding CLIF:

```
(exists ((x Number) (y Number))
   (and (Number 7) (= y (Divide (Add x 7) (Sqrt 7)))))
```

No CLIF variables are needed to represent the coreference labels `*u` and `*v` since the functional notation used in CLIF shows the connections directly.

CLIF only permits functions to have a single output, but extended CGIF allows actors to have multiple outputs. The following actor of type `IntegerDivide` has two inputs: an integer `x` and an integer `7`. It also has two outputs: a quotient `u` and a remainder `v`.

```
(IntegerDivide [Integer: *x] [Integer: 7] | [*u] [*v])
```

When this actor is translated to core CGIF or CLIF, the vertical bar is removed, and the actor becomes an ordinary relation with four arguments; the distinction between inputs and outputs is lost. In order to assert the constraint that the last two arguments are functionally dependent on the first two arguments, the following CGIF sentence asserts that there exist two functions, identified by the coreference labels

Quotient and Remainder, which for every combination of input and output values are logically equivalent to an actor of type IntegerDivide with the same input and output values:

```
[Function: *Quotient] [Function: *Remainder]
[[@every*x1] [@every*x2] [@every*x3] [@every*x4]
[Equiv: [Iff: (IntegerDivide ?x1 ?x2 | ?x3 ?x4)]
        [Iff: (#?Quotient ?x1 ?x2 | ?x3) (#?Remainder ?x1 ?x2 | ?x4)]]]
```

Each line of this example illustrates one or more features of CGIF. The first line represents existential quantifiers for two entities of type Function. On the second line, the context bracket [encloses the concept nodes with universal quantifiers, marked by @every, to show that the existential quantifiers for Quotient and Remainder include the universals within their scope. The equivalence on lines three and four shows that an actor of type IntegerDivide is logically equivalent to a conjunction of the quotient and remainder functions. Finally, the symbol # on line four shows that the coreference labels ?Quotient and ?Remainder are being used as type labels. Following is the corresponding CLIF:

```
(exists ((Quotient Function) (Remainder Function))
   (forall (x1 x2 x3 x4)
      (iff (IntegerDivide x1 x2 x3 x4)
           (and (= x3 (Quotient x1 x2)) (= x4 (Remainder x1 x2))))))
```

As another example of the use of quantification over relations, someone might say "Bob and Sue are related," but not say exactly how they are related. The following sentences in CGIF and CLIF state that there exists some familial relation *r* that relates Bob and Sue:

```
[Relation: *r] (Familial ?r) (#?r Bob Sue)

(exists ((r Relation)) (and (Familial r) (r Bob Sue)))
```

The concept [Relation: *r] states that there exists a relation *r*. The next two relations state that *r* is familial and *r* relates Bob and Sue.

This brief survey has illustrated nearly every major feature of CGIF and CLIF. One important feature that has not been mentioned is the use of *sequence variables* to support relations with a variable number of arguments. Another is the use of comments, which can be placed before, after, or inside any concept or relation node in CGIF. The specifications in the CL standard guarantee that any sentence expressed in any of the three fully conformant dialects — CLIF, CGIF, or XCL — can be translated to any of the others in a logically equivalent form. Although the translation will preserve the semantics, it is not guaranteed to preserve all syntactic details: a sentence translated from one dialect to another and then back to the first will be logically equivalent to the original, but some subexpressions might be reordered or replaced by semantic equivalents.

In general, Common Logic is a superset of many different logic-based languages and notations, including the traditional predicate-calculus notation for first-order logic. But since various languages have been designed and implemented at widely separated times and places, that generalization must be qualified with different caveats for each case:

- **Semantic Web Languages**. The CL standard supports the URIs defined by the W3C as valid CL name sequences, and it allows text stored on the web to be imported into CLIF, CGIF, or XCL documents. The tools that import the text could, if necessary, translate one dialect to another at import time. Since the semantics for RDF(S) and OWL was designed as a subset of the CL model theory, those languages can be translated to any fully conformant CL dialect

(Hayes 2005).

- **Z Specification Language**. The Z model theory is a subset of the CL model theory, but the syntax of Z enforces strong type checking, and it does not permit quantifiers to range over functions and relations. Therefore, Z statements can be translated to CL, but only those statements that originally came from Z are guaranteed to be translatable back to Z.
- **Unified Modeling Language (UML)**. Although the UML diagrams and notations are loosely based on logic, they have no formal specification in any version of logic. The best hope for providing a reliable foundation for UML would be to implement tools that translate UML to CL. If done properly, such tools could define a *de facto* standard for UML semantics.
- **Logic-Programming Languages**. Well-behaved languages that support classical negation can be translated to CL while preserving the semantics. Languages such as Prolog that are based on negation as failure could be translated to CL, but with the usual caveats about ways of working around the discrepancies.
- **SQL Database Language**. The WHERE clause in SQL queries and constraints can state an arbitrary FOL expression, but problems arise with the treatment of null values in the database and with differences between the open-world and closed-world assumptions. To avoid the nonlogical features of SQL, CL can be mapped to and from the Datalog language, which supports the Horn-clause subset of FOL and has a direct mapping to the SQL operations.

Most people have strong attachments to their favorite syntactic features. The goal of the Common Logic project is to provide a very general semantics that enables interoperability at the semantic level despite the inevitable syntactic differences. CL has demonstrated that such seemingly diverse notations as conceptual graphs, predicate calculus, and the languages of the Semantic Web can be treated as dialects with a common semantic foundation. An extension of CL called IKL, which is discussed in Section 5, can support an even wider range of logics.

3. Reasoning with Graphs

Graphs have some advantages over linear notations in both human factors and computational efficiency. As Figures 1 to 8 illustrate, graphs show relationships at a glance that are harder to see in linear notations, including CGIF and CLIF. Graphs also have a highly regular structure that can simplify many algorithms for reasoning, searching, indexing, and pattern matching. Yet AI research has largely ignored the structural properties of graphs, and some of the most advanced research on representing, indexing, and manipulating graphs has been done in organic chemistry. With his BS degree in chemistry, Peirce was the first to recognize the similarity between chemical graphs and logical graphs. He wanted to represent the "atoms and molecules of logic" in his existential graphs, and he used the word *valence* for the number of arguments of a relation. By applying algorithms for chemical graphs to conceptual graphs, Levinson and Ellis (1992) implemented the first type hierarchy that could support retrieval and classification in logarithmic time. More recent research on chemical graphs has been used in algorithms for computing semantic distance between CGs. Those techniques have enabled analogy finding in logarithmic time, instead of the polynomial-time computations of the older methods (Sowa & Majumdar 2003).

The six *canonical formation rules* (Sowa 2000) are examples of graph-based operators that focus on the semantics. Combinations of these rules, called *projection* and *maximal join*, perform larger semantic operations, such as answering a question or comparing the relevance of different alternatives. Each rule has one of three possible effects on the logical relationship between a starting graph *u* and the resulting graph *v*:

- **Equivalence**. *Copy* and *simplify* are equivalence rules, which generate a graph *v* that is logically equivalent to the original: $u \supset v$ and $v \supset u$. Equivalent graphs are true in exactly the same models.
- **Specialization**. *Join* and *restrict* are specialization rules, which generate a graph *v* that implies the original: $v \supset u$. Specialization rules monotonically decrease the set of models in which the result is true.
- **Generalization**. *Detach* and *unrestrict* are generalization rules, which generate a graph *v* that is implied by the original: $u \supset v$. Generalization rules monotonically increase the set of models in which the result is true.

Each rule has an inverse rule that undoes any change caused by the other. The inverse of copy is simplify, the inverse of restrict is unrestrict, and the inverse of join is detach. These rules are fundamentally graphical: they are easier to show than to describe. The next three diagrams (Figures 9, 10, and 11) illustrate these rules with *simple graphs*, which use only conjunction and existential quantifiers. When rules for handling negation are added, they form a complete proof procedure for first-order logic with equality.

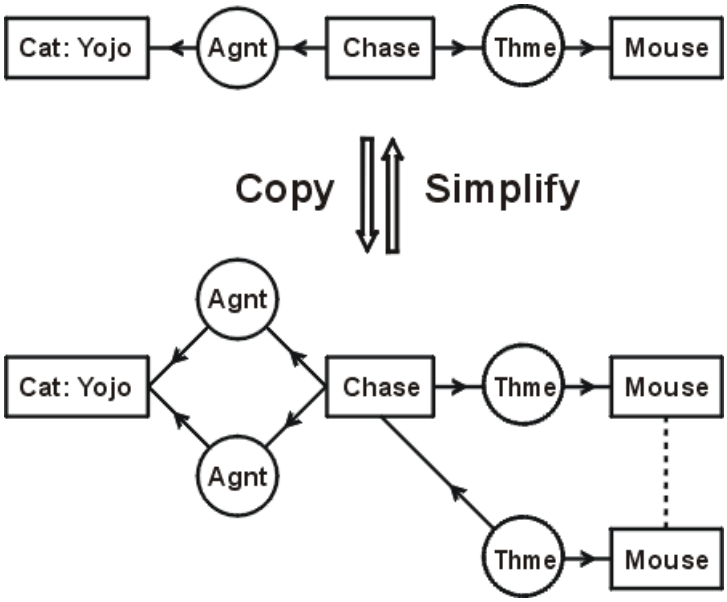

Figure 9: Copy and simplify rules

The CG at the top of Figure 9 represents the sentence *The cat Yojo is chasing a mouse*. The down arrow represents two applications of the copy rule. One application copies the `Agnt` relation, and a second copies the subgraph `→(Thme)→[Mouse]`. The coreference link connecting the two `[Mouse]` concepts indicates that both concepts refer to the same individual. The up arrow represents two applications of the simplify rule, which performs the inverse operations of erasing redundant copies. Following are the CGIF sentences for both graphs:

```
[Cat: Yojo] [Chase: *x] [Mouse: *y] (Agent ?x Yojo) (Thme ?x ?y)

[Cat: Yojo] [Chase: *x] [Mouse: *y] [Mouse: ?y]
(Agent ?x Yojo) (Agent ?x Yojo) (Thme ?x ?y) (Thme ?x ?y)
```

As the CGIF illustrates, the copy rule makes redundant copies, which are erased by the simplify rule. In effect, the copy rule is $p \supset (p \wedge p)$, and the simplify rule is $(p \wedge p) \supset p$.

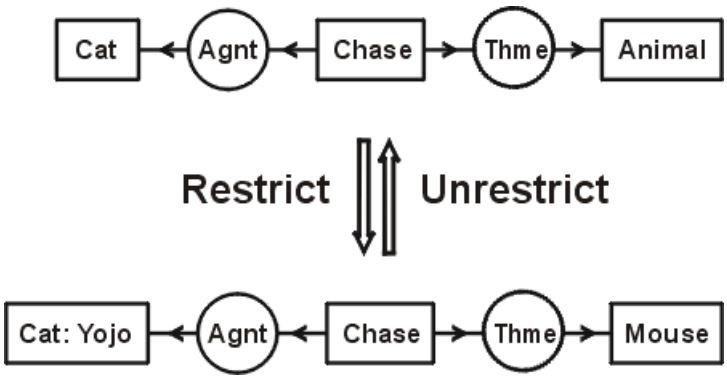

Figure 10: Restrict and unrestrict rules

The CG at the top of Figure 10 represents the sentence *A cat is chasing an animal*. By two applications of the restrict rule, it is transformed to the CG for *The cat Yojo is chasing a mouse*. In the first step, the concept `[Cat]`, which says that there exists some cat, is *restricted by referent* to the more specific concept `[Cat: Yojo]`, which says that there exists a cat named Yojo. In the second step, the concept `[Animal]`, which says that there exists an animal, is *restricted by type* to a concept of a subtype `[Mouse]`. The more specialized graph implies the more general one: if the cat Yojo is chasing a mouse, then a cat is chasing an animal.

To show that the bottom graph v of Figure 10 implies the top graph u, let c be a concept of u that is being restricted to a more specialized concept d, and let u be $c \wedge w$, where w is the remaining information in u. By hypothesis, $d \supset c$. Therefore, $(d \wedge w) \supset (c \wedge w)$. Hence, $v \supset u$.

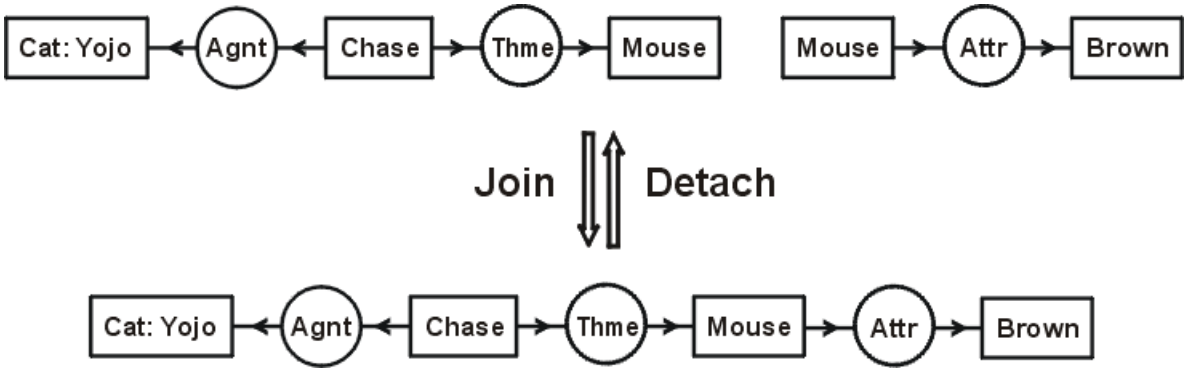

Figure 11: Join and detach rules

At the top of Figure 11 are two CGs for the sentences *Yojo is chasing a mouse* and *A mouse is brown*. The join rule overlays the two identical copies of the concept `[Mouse]` to form a single CG for the sentence *Yojo is chasing a brown mouse*. The detach rule undoes the join to restore the top graphs. Following are the CGIF sentences that represent the top and bottom graphs of Figure 11:

```
[Cat: Yojo] [Chase: *x] [Mouse: *y] (Agent ?x Yojo) (Thme ?x ?y)
[Mouse: *z] [Brown: *w] (Attr ?z ?w)

[Cat: Yojo] [Chase: *x] [Mouse: *y] (Agent ?x Yojo) (Thme ?x ?y)
[Brown: *w] (Attr ?y ?w)
```

As the CGIF illustrates, the bottom graph consists of substituting `y` for every occurrence of `z` in the top graph and erasing redundant copies. In general, every join assumes an equality of the form $y=z$ and simplifies the result. If q is the equality and u is original pair of graphs at the top, then the bottom graph

is equivalent to $q \wedge u$, which implies u. Therefore, the result of join implies the original graphs.

Together, the generalization and equivalence rules are sufficient for a complete proof procedure for the subset of logic whose only operators are conjunction and the existential quantifier. The specialization and equivalence rules can be used in a refutation procedure for a proof by contradiction. To handle full first-order logic, rules for negations must be added. Peirce defined a complete proof procedure for FOL whose rules depend on whether a context is positive (nested in an even number of negations, possibly zero) or negative (nested in an odd number of negations). Those rules are grouped in three pairs, one of which (i) inserts a graph and the other (e) erases a graph. The only axiom is a blank sheet of paper (an empty graph with no nodes); in effect, the blank is a generalization of all other graphs. Following is a restatement of Peirce's rules in terms of specialization and generalization. These same rules apply to both propositional logic and full first-order logic. In FOL, the operation of inserting or erasing a connection between two nodes has the effect of identifying two nodes (i.e., a substitution of a value for a variable) or treating them as possibly distinct.

1. i. In a negative context, any graph or subgraph (including the blank) may be replaced by any specialization.

   e. In a positive context, any graph or subgraph may be replaced by any generalization (including the blank).

2. i. Any graph or subgraph in any context *c* may be copied in the same context *c* or into any context nested in *c*. (The only exception is that no graph may be copied directly into itself; however, it is permissible to copy a graph *g* in the context *c* and then to copy the copy into the original *g*.)

   e. Any graph or subgraph that could have been derived by rule 2i may be erased. (Whether or not the graph was in fact derived by 2i is irrelevant.)

3. i. A double negation (nest of two negations with nothing between the inner and outer) may be drawn around any graph, subgraph, or set of graphs in any context.

   e. Any double negation in any context may be erased.

This version of the rules was adapted from a tutorial on existential graphs by Peirce (1909). He originally formulated these rules in 1897 and finally published them in 1906, but they are a simplification and generalization of the rules of *natural deduction* by Gentzen (1935). When these rules are applied to CGIF, some adjustments may be needed to rename coreference labels or to convert a bound label to a defining label or vice versa. For example, if a defining node is erased, a bound label may become the new defining label. Such adjustments are not needed in the pure graph notation. For further discussion of Peirce's rules of inference, see the commentary by Sowa (2010).

All the axioms and rules of inference for classical FOL, including the rules of the *Principia Mathematica*, natural deduction, and resolution, can be proved in terms of Peirce's rules. As an example, Frege's first axiom, written in Peirce-Peano notation, is $a \supset (b \supset a)$. Figure 12 shows a proof by Peirce's rules. To improve contrast, positive areas are shown in white, and negative areas are shaded.

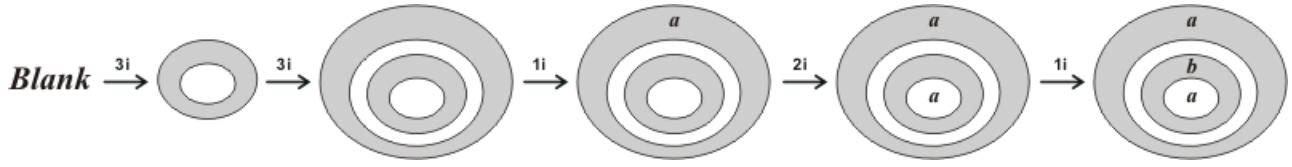

Figure 12: Proof of Frege's first axiom by Peirce's rules

In core CGIF, the propositions *a* and *b* could be represented as relations with zero arguments. Following are the five steps of Figure 12 written in core CGIF:

1. By rule 3i, Insert a double negation around the blank: `~[ ~[ ]]`
2. By 3i, insert a double negation around the previous one: `~[ ~[ ~[ ~[ ]]]]`
3. By 1i, insert `(a)`: `~[ (a) ~[ ~[ ~[ ]]]]`
4. By 2i, copy `(a)`: `~[ (a) ~[ ~[ ~[ (a) ]]]]`
5. By 1i, insert `(b)`: `~[ (a) ~[ ~[ (b) ~[ (a) ]]]]`

The theorem to be proved contains five symbols, and each step of the proof inserts one symbol into its proper place in the final result. Frege had a total of eight axioms, and the *Principia* had five. All of them could be derived by similarly short proofs.

Frege's two rules of inference, which Whitehead and Russell adopted, were *modus ponens* and *universal instantiation*. Figure 13 is a proof of *modus ponens*, which derives *q* from a statement *p* and an implication $p \supset q$:

Figure 13: Proof of modus ponens

Following are the four EGs of Figure 13 written in core CGIF:

1. Starting graphs: `(p) ~[ (p) ~[ (q) ]]`
2. By 2e, erase the nested copy of `(p)`: `(p) ~[ ~[ (q) ]]`
3. By 1e, erase `(p)`: `~[ ~[ (q) ]]`
4. By 3e, erase the double negation: `(q)`

Frege's other rule of inference is *universal instantiation*, which allows any term *t* to be substituted for a universally quantified variable in a statement of the form $(\forall x)P(x)$. In EGs, the term *t* would be represented by a graph of the form —*t*, which states that something satisfying the condition *t* exists, and the universal quantifier corresponds to a negated existential: a line whose outermost part (the existential quantifier) occurs in a negative area. Since a graph has no variables, there is no notion of substitution. Instead, the proof in Figure 14 performs the equivalent operation by connecting the two lines.

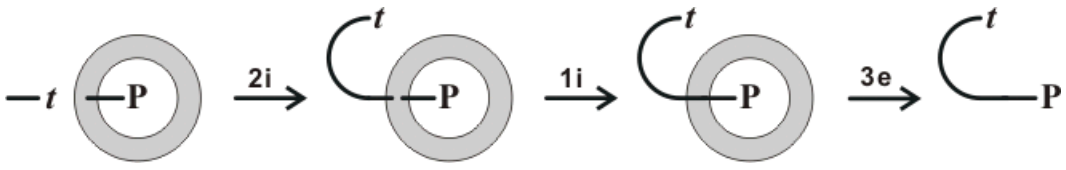

Figure 14: Proof of universal instantiation

The absence of labels on the EG lines simplifies the proofs by eliminating the need to relabel variables in CLIF or coreference links in CGIF. In core CGIF, the first step is the linear version of Figure 14:

1. Starting graphs: `[*x] (t ?x) ~[ [*y] ~[ (P ?y) ]]`
2. By 2i, copy `[*x]` and change the defining label `*x` to a bound label `?x` in the copy:

```
[*x] (t ?x) ~[ [?x] [*y] ~[ (P ?y) ]]
```

3. By 1i, insert a connection between the two lines. In CGIF, that corresponds to relabeling `*y` and `?y` to `?x` and erasing redundant copies of `[?x]`: `[*x] (t ?x) ~[ ~[ (P ?x) ]]`
4. By 3e, erase the double negation: `[*x] (t ?x) (P ?x)`

With the universal quantifier in extended CGIF, the starting graphs of Figure 14 could be written

```
[*x] (t ?x) [(P [@every*y])]
```

The extra brackets around the last node ensure that the existential quantifier `[*x]` includes the universal `@every*y` within its scope. Then universal instantiation could be used as a one-step derived rule to generate line 4. After `@every` has been erased, the brackets around the last node are not needed and may be erased.

In the *Principia Mathematica*, Whitehead and Russell proved the following theorem, which Leibniz called the *Praeclarum Theorema* (Splendid Theorem). It is one of the last and most complex theorems in propositional logic in the *Principia*, and it required a total of 43 steps, starting from five nonobvious axiom schemata.

$$((p \supset r) \wedge (q \supset s)) \supset ((p \wedge q) \supset (r \wedge s))$$

With Peirce's rules, this theorem can be proved in just seven steps starting with a blank sheet of paper (Figure 15). Each step of the proof inserts or erases one graph, and the final graph is the statement of the theorem.

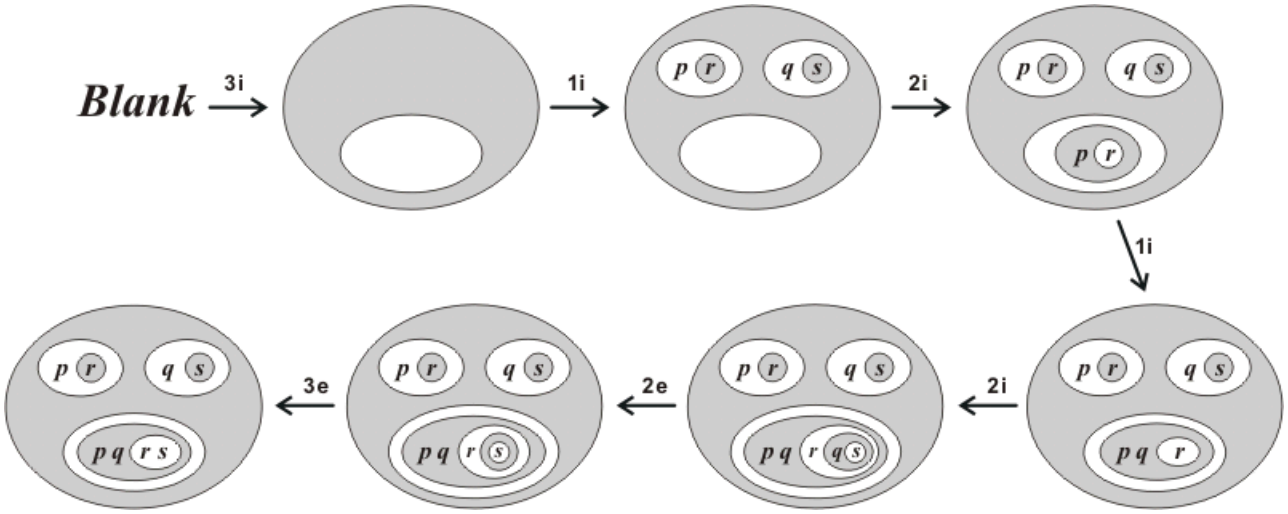

Figure 15: Proof in 7 steps instead of 43 in the *Principia*

After only four steps, the graph looks almost like the desired conclusion, except for a missing copy of *s* inside the innermost area. Since that area is positive, it is not permissible to insert *s* directly. Instead, Rule 2i copies the graph that represents $q \supset s$. By Rule 2e, the next step erases an unwanted copy of *q*. Finally, Rule 3e erases a double negation to derive the conclusion.

Unlike Gentzen's version of natural deduction, which uses a method of making and discharging assumptions, Peirce's proofs proceed in a straight line from a blank sheet to the conclusion: every step inserts or erases one subgraph in the immediately preceding graph. As Figure 15 illustrates, the first two steps of any proof that starts with a blank must draw a double negation around the blank and insert

a graph into the negative area. That graph is usually the entire hypothesis of the theorem to be proved. The remainder of the proof develops the conclusion in the doubly nested blank area. Those two steps are the equivalent of Gentzen's method of making and discharging an assumption, but in Gentzen's approach, the two steps may be separated by arbitrarily many intervening steps, and a system of bookkeeping is necessary to keep track of the assumptions. With Peirce's rules, the second step follows immediately after the first, and no bookkeeping is required.

Most common proofs take about the same number of steps with Peirce's rules as with Gentzen's version of natural deduction or his *sequent calculus*. For some kinds of proofs, however, Peirce's rules can be much faster because of a property that is not shared by any other common proof procedure: the rules depend only on whether an area is positive or negative; the depth of nesting is irrelevant. That property implies the "cut-and-paste theorem" (Sowa 2000), which is proved in terms of Peirce's rules, but it can be used in any notation for first-order logic:

- **Theorem.** If a proof $p \vdash q$ is possible on a blank sheet, then in any positive area of a graph or formula where p occurs, q may be substituted for p.
- **Proof.** Since the area in which p occurs is positive, every step of the proof of q can be carried out in that area. Therefore, it is permissible to "cut out" and "paste in" the steps of the proof from p to q in that area. After q has been derived, Rule 1e can be applied to erase the original p and any remaining steps of the proof other than q.

Dau (2006) showed that certain proofs that take advantage of this theorem or the features of Peirce's rules that support it can be orders of magnitude shorter than proofs based on other rules of inference. Conventional rules, for example, can only be applied to the outermost operator. If a graph or formula happens to contain a deeply nested subformula p, those rules cannot replace it with q. Instead, many steps may be needed to bring p to the surface of some formula to which conventional rules can be applied. An example is the *cut-free* version of Gentzen's sequent calculus, in which proofs can sometimes be exponentially longer than proofs in the usual version. With Peirce's rules, the corresponding cut-free proofs are only longer by a polynomial factor.

The canonical formation rules have been implemented in nearly all CG systems, and they have been used in formal logic-based methods, informal case-based reasoning, and various computational methods. A multistep combination, called a *maximal join*, is used to determine the extent of the unifiable overlap between two CGs. In natural language processing, maximal joins can help resolve ambiguities and determine the most likely connections of new information to background knowledge and the previous context of a discourse. Stewart (1996) implemented Peirce's rules of inference in a first-order theorem prover for EGs and showed that its performance is comparable to resolution theorem provers. In all reasoning methods, formal and informal, a major part of the time is spent in searching for relevant rules, axioms, or background information. Ongoing research on efficient methods of indexing graphs and selecting the most relevant information has shown great improvement in many cases, but more work is needed to incorporate such indexing into conventional reasoning systems.

4. Propositions, Situations, and Metalanguage

Natural languages are highly expressive systems that can state anything that has ever been stated in any formal language or logic. They can even express metalevel statements about themselves, their relationships to other languages, and the truth of any such statements. Such enormous expressive power can easily generate contradictions and paradoxes, such as the statement *This sentence is false*. Most formal languages avoid such paradoxes by imposing restrictions on the expressive power. Common

Logic, for example, can represent any sentence in any CL dialect as a quoted string, and it can even specify the syntactic structure of such strings. But CL has no mechanism for treating such strings as CL sentences and relating substrings in them to the corresponding CL names.

Although the paradoxes of logic are expressible in natural language, the most common use of language about language is to talk about the beliefs, desires, and intentions of the speaker and other people. Many versions of logic and knowledge representation languages, including conceptual graphs, have been used to express such language. As an example, the sentence *Tom believes that Mary wants to marry a sailor*, contains three clauses, whose nesting may be marked by brackets:

```
Tom believes that [Mary wants [to marry a sailor]].
```

The outer clause asserts that Tom has a belief, which is expressed by the object of the verb *believe*. Tom's belief is that Mary wants a situation described by the nested infinitive, whose subject is the same person who wants the situation. Each clause makes a comment about the clause or clauses nested in it. References to the individuals mentioned in those clauses may cross context boundaries in various ways, as in the following two interpretations of the original English sentence:

```
Tom believes that [there is a sailor whom Mary wants [to marry]].
There is a sailor whom Tom believes that [Mary wants [to marry]].
```

The two conceptual graphs in Figure 16 represent the first and third interpretations. In the CG on the left, the existential quantifier for the concept `[Sailor]` is nested inside the situation that Mary wants. Whether such a sailor actually exists and whether Tom or Mary knows his identity are undetermined. The CG on the right explicitly states that such a sailor exists; the connections of contexts and relations imply that Tom knows him and that Tom believes that Mary also knows him. Another option (not shown) would place the concept `[Sailor]` inside the context of type `Proposition`; it would leave the sailor's existence undetermined, but it would imply that Tom believes he exists and that Tom believes Mary knows him.

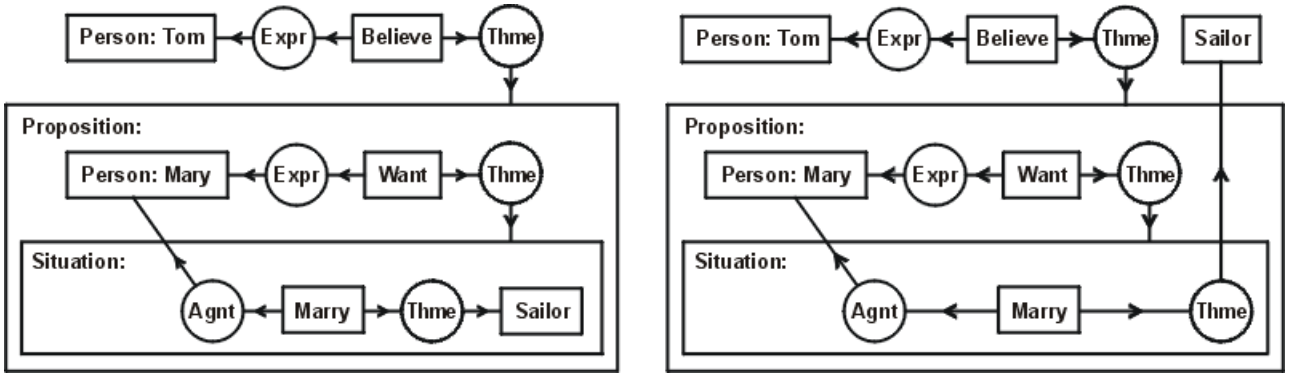

Figure 16: Two interpretations of *Tom believes that Mary wants to marry a sailor*

The context boxes illustrated in Figures 4 and 6 express negations or operators such as `If` and `Then`, which are defined in terms of negations. But the contexts of type `Proposition` and `Situation` in Figure 16 raise new issues of logic and ontology. The CL semantics can represent entities of any type, including propositions and situations, but it has no provision for relating such entities to the internal structure of CL sentences. A more expressive language, called IKL (Hayes & Menzel 2006), was defined as an upward compatible extension of CL. The IKL semantics introduces entities called *propositions* and a special operator, spelled `that`, which relates IKL sentences to the propositions they express. IKL semantics does not have a built-in type for situations, but it is possible in IKL to make statements that state the existence of entities of type `Situation` and relate them to propositions.

The first step toward translating the CGs in Figure 16 to IKL is to write them in an extended version of CGIF, which allows CGs to be nested inside concept nodes of type `Proposition` or `Situation`. Following is the CGIF for the CG on the left:

```
[Person: Tom] [Believe: *x1] (Expr ?x1 Tom)
(Thme ?x1 [Proposition:
   [Person: Mary] [Want: *x2] (Expr ?x2 Mary)
   (Thme ?x2 [Situation:
      [Marry: *x3] [Sailor: *x4] (Agnt ?x3 Mary) (Thme ?x3 ?x4)])])
```

This statement uses the option of moving the concept nodes for the types `Proposition` and `Situation` inside the relation nodes of type `Thme`. That option has no semantic significance, but it makes the order of writing the CGIF closer to English word order. A much more important semantic question is the relation between situations and propositions. In the ontology commonly used with CGs, that relation is spelled `Dscr` and called the *description relation*. The last two lines of the CGIF statement above could be rewritten in the following form:

```
(Thme ?x2 [Situation: *s]) (Dscr ?s [Proposition:
   [Marry: *x3] [Sailor: *x4] (Agnt ?x3 Mary) (Thme ?x3 ?x4)])])
```

The last line is unchanged, but the line before it states that the theme of `x2` is the situation `s` and the description of `s` is the proposition stated on the last line. In effect, every concept of type `Situation` that contains a nested CG is an abbreviation for a situation that is described by a concept of type `Proposition` that has the same nested CG. This expanded CGIF statement can then be translated to IKL (which is based on CLIF syntax with the addition of the operator `that`).

```
(exists ((x1 Believe)) (and (Person Tom) (Expr x1 Tom)
(Thme x1 (that
   (exists ((x2 Want) (s Situation)) (and (Person Mary) (Expr x2 Mary)
   (Thme x2 s) (Dscr s (that
      (exists ((x3 Marry) (x4 Sailor)) (and (Agnt x3 Mary) (Thme x3 x4)
))))))))))
```

Each line of the IKL statement expresses the equivalent of the corresponding line in CGIF. Note that every occurrence of `Proposition` in CGIF corresponds to `that` in IKL. The syntax of CLIF or IKL requires more parentheses than CGIF because every occurrence of `(exists` or `(and` requires an extra closing parenthesis at the end.

As these examples illustrate, the operator `that` adds an enormous amount of expressive power, but IKL still has a first-order style of semantics. The proposition nodes in CGs or the `that` operator in IKL introduce abstract entities of type `Proposition`. Although language about propositions is a kind of metalanguage, it does not, by itself, go beyond first-order logic. Tarski (1933), for example, demonstrated how a stratified series of metalevels, each of which is purely first order, can be used without creating paradoxes or going beyond the semantics of FOL. In effect, Tarski avoided paradoxes by declaring that certain kinds of sentences (those that violate the stratification) do not express propositions in his models. The IKL model theory avoids the paradoxes by deriving a stable, but false truth value for the troublesome sentences. For example, the following English sentence, which sounds paradoxical, could be expressed in either IKL or CGIF syntax:

There exists a proposition p, p is true, and p says that p is false.

Following is a translation to IKL:

```
(exists ((p Proposition)) (and (p) (= p (that (not p)))))
```

Literally, the IKL sentence says that there exists a proposition `p`, it asserts `(p)`, and it says that `p` is identical to the proposition that `(not p)`. It is false because no proposition is identical to its negation. Following is the equivalent in CGIF:

```
[Proposition: *p] (#?p) [Proposition: ?p ~[(#?p)]]
```

The first concept says that there exists a proposition `p`, the relation `(#?p)` asserts `p`, and the last concept says that `p` is coreferent with the proposition that not `p`. Coreference in CGIF has the same effect as equality in CLIF.

5. Research Extensions

Over the years, the term *conceptual graph* has been used in a broad sense as any notation that has a large overlap with the the notation used in the book by Sowa (1984). That usage has led to a large number of dialects with varying degrees of compatibility. The purpose of a standard is to stabilize a design at a stage where it can provide a fixed, reliable platform for the development of products and applications. A fixed design, however, is an obstacle to innovation in the platform itself, although it is valuable for promoting innovation in applications that use the platform. In order to support fundamental research while providing a stable platform for applications, it's important to distinguish ISO standard CGs, IKL CGs, and research CGs. The first two provide rich and stable platforms for application development, while the third allows research projects to add extensions and modifications, which may be needed for a particular application and which may someday be added to the standard.

Most of the features of the research CGs are required to support natural language semantics. Some of them, such as modal operators, are as old as Aristotle, but they are not in the CL standard because their semantics involves open research issues. Following are the most common research extensions that have been proposed or implemented in various forms over the years:

- **Contexts.** Peirce's first use for the oval was to negate the graphs nested inside, and that is the only use that is recognized by the CL standard. But Peirce (1898) generalized the ovals to context enclosures, which allow relations other than negation to be linked to a context. Most of those features could be defined in terms of the IKL extensions described in Section 4, but there is no consensus on any definitions that could be considered for a standard.
- **Metalanguage.** The basic use of a context enclosure is to quote the nested graphs. That metalevel syntax allows any semantic approach to be defined by axioms that specify how the nested graphs are interpreted. Sowa (2003, 2006) adapted that approach to a family of *nested graph models* (NGMs), which could be used to formalize the semantics of many kinds of modal and intensional logics. A hierarchy of metalevels with the NGM semantics can express the equivalent of a wide range of modal, temporal, and intentional logics. The most useful NGMs can be represented with the IKL semantics, but the many variations and their application to natural languages have not yet been fully explored.
- **Generalized quantifiers.** In addition to the usual quantifiers of *every* and *some*, natural languages support an open-ended number of quantificational expressions, such as *exactly one*, *at least seven*, or *considerably more*. Some of these quantifiers, such as *exactly one cat*, could be represented as `[Cat: @1]` and defined in terms of the CL standard. Others, such as *at least seven cats*, could be represented `[Cat: @≤7]` and defined with a version of set theory added to the base logic. But quantifiers such as *considerably more* would require some method of approximate reasoning, such as fuzzy sets or rough sets.

- **Indexicals.** Peirce observed that every statement in logic requires at least one indexical to fix the referents of its symbols. The basic indexical, which corresponds to the definite article *the*, is represented by the symbol # inside a concept node: `[Dog: #]` would represent the phrase *the dog*. The pronouns *I*, *you*, and *she* would be represented `[Person: #I]`, `[Person: #you]`, and `[Person: #she]`. To process indexicals, some linguists propose versions of *dynamic semantics*, in which the model is updated during the discourse. A simpler method is to treat the # symbol as a syntactic marker that indicates a incomplete interpretation of the original sentence. With this approach, the truth value of a CG that contains any occurrences of # is not determined until those markers are replaced by names or coreference labels. This approach supports indexicals in an intermediate representation, but uses a conventional model theory to evaluate the final resolution.
- **Plural nouns.** Plurals have been represented in CGs by set expressions inside the concept boxes. The concept `[Cat: {*}@3]` would represent *three cats*, and `[Dog: {Lucky, Macula}]` would represent *the dogs Lucky and Macula*. Various methods have been proposed for representing distributed and collective plurals and translating them to versions of set theory and mereology. But the representation of plurals is still a research area in linguistics, and it is premature to adopt a standard syntax or semantics.
- **Procedural attachments.** The CL standard defines actors as purely functional relations, but various implementations have allowed more informal versions, in which the actors represent arbitrary procedures. Some versions have implemented *token passing* algorithms with the symbol `?` for a backward-chaining request used in a *query graph*, and with the symbol `!` for a forward-chaining trigger that asserts a new value. As examples, the concept `[Employee: ?]` would ask *which employee*, and the concept `[Employee: John!]` would assert *the employee John*.

As an example of applied research, one of the largest commercial CG systems is Sonetto (Sarraf & Ellis 2006), which uses extended versions of the earlier algorithms by Levinson and Ellis (1992). A key innovation of Sonetto is its semi-automated methods for extracting ontologies and business rules from unstructured documents. The users who assist Sonetto in the knowledge extraction process are familiar with the subject matter, but they have no training in programming or knowledge engineering. CGIF is the knowledge representation language for ontologies, rules, and queries. It is also used to manage the schemas of documents and other objects in the system and to represent the rules that translate CGIF to XML and other formats. For CG research and applications, see the collections edited by Nagle et al. (1992), Way (1992), Chein (1996), and Schärfe and Hitzler (2009). Ongoing developments of conceptual graphs and related systems are published in the annual proceedings of the International Conferences on Conceptual Structures.

Appendix: CGIF Grammar

Lexical Grammar Rules

The syntax rules are written in Extended Backus-Naur Form (EBNF) rules, as specified by ISO/IEC 14977. The CGIF syntax rules assume the same four types of names as CLIF: `namecharsequence` for names not enclosed in quotes; `enclosedname` for names enclosed in double quotes; `numeral` for numerals consisting of one or more digits; and `quotedstring` for character strings enclosed in single quotes. But because of syntactic differences between CGIF and CLIF, CGIF must enclose more names in quotes than CLIF in order to avoid ambiguity. Therefore, the only CG names not enclosed in

quotes belong to the categories `identifier` and `numeral`.

```
CGname = identifier | '"', (namecharsequence - identifier), '"'
         | numeral | enclosedname | quotedstring;
identifier = letter, {letter | digit | "_"};
```

When CGIF is translated to CL, a CGname is translated to a CLIF name by removing any quotes around a name character sequence. CLIF does not make a syntactic distinction between constants and variables, but in CGIF any CGname that is not used as a defining label or a bound label is called a *constant*. The start symbol of CGIF syntax is the category `text` for a complete text or the category `CG` for just a single conceptual graph.

Core CGIF Grammar Rules

An *actor* is a conceptual relation that represents a function in Common Logic. It begins with `(`, an optional comment, an optional string `#?`, a CG name, `|`, an arc, an optional end comment, and `)`. If the CG name is preceded by `#?`, it represents a bound coreference label; otherwise, it represents a type label. The arc sequence represents the arguments of the CL function and the last arc represents the value of the function.

```
actor = "(", [comment], ["#?"], CGname, arcSequence, "|", arc,
             [endComment], ")";
```

An *arc* is an optional comment followed by a reference. It links an actor or a conceptual relation to a concept that represents one argument of a CL function or relation.

```
arc = [comment], reference;
```

An *arc sequence* is a sequence of zero or more arcs, followed by an option consisting of an optional comment, `?`, and a sequence marker.

```
arcSequence = {arc}, [[comment], "?", seqmark];
```

A *comment* or an *end comment* is a character string that has no effect on the semantics of a conceptual graph or any part of a conceptual graph. A comment begins with `"/*"`, followed by a character string that contains no occurrence of `"*/"`, and ends with `"*/"`. A comment may occur immediately after the opening bracket of any concept, immediately after the opening parenthesis of any actor or conceptual relation, immediately before any arc, or intermixed with the concepts and conceptual relations of a conceptual graph. An end comment begins with `;`, followed by a character string that contains no occurrence of `]` or `)`. An end comment may occur immediately before the closing bracket of any concept or immediately before the closing parenthesis of any actor or conceptual relation.

```
comment = "/*", {(character-"*") | ["*", (character-"/")]}, ["*"], "*/";

endComment = ";", {character - ("]" | ")")};
```

A *concept* is either a context, an existential concept, or a coreference concept. Every concept begins with `[` and an optional comment; and every concept ends with an optional end comment and `]`. Between the beginning and end, a context contains a CG; an existential concept contains `*` and either a CG name or a sequence marker; and a coreference concept contains `:` and a sequence of one or more references. A context that contains a blank CG is said to be *empty*, even if it contains one or more comments; any comment that occurs immediately after the opening bracket shall be part of the concept,

not the following CG.

```
concept = "[", [comment],
           (CG | "*", (CGname | seqmark) | ":", {reference}- ),
           [endComment], "]";
```

A *conceptual graph* (CG) is an unordered list of concepts, conceptual relations, negations, and comments.

```
CG = {concept | conceptualRelation | negation | comment};
```

A *conceptual relation* is either an ordinary relation or an actor. An ordinary relation, which represents a CL relation, begins with `(`, an optional comment, an optional string `#?`, a CG name, an optional end comment, and `)`. If the CG name is preceded by `#?`, it represents a bound coreference label; otherwise, it represents a type label. An ordinary relation has just one sequence of arcs, but an actor has two sequences of arcs.

```
conceptualRelation = ordinaryRelation | actor;

ordinaryRelation = "(", [comment], ["#?"], CGname, arcSequence,
                         [endComment], ")";
```

A *negation* is ~ followed by a context.

```
negation = "~", context;
```

A *reference* is an optional `?` followed by a CG name. A CG name prefixed with `?` is called a *bound coreference label*; without the prefix `?`, it is called a *constant*.

```
reference = ["?"], CGname;
```

A *text* is a context, called an *outermost context*, that has an optional name, has an arbitrarily large conceptual graph, and is not nested inside any other context. It consists of `[`, an optional comment, the type label `Proposition`, `:`, an optional CG name, a conceptual graph, an optional end comment, and `]`. Although a text may contain core CGIF, the type label Proposition is outside the syntax of core CGIF.

```
text = "[", [comment], "Proposition", ":", [CGname], CG,
            [endComment], "]";
```

Extended CGIF Grammar Rules

Extended CGIF is superset of core CGIF, and every syntactically correct sentence of core CGIF is also syntactically correct in extended CGIF. Its most prominent feature is the option of a *type label* or a *type expression* on the left side of any concept. In addition to types, extended CGIF adds the following features to core CGIF:

- More options in concepts, including universal quantifiers.
- Boolean contexts for representing the operators or, if, and iff.
- The option of allowing concept nodes to be placed in the arc sequence of conceptual relations.
- The ability to import text into a text.

These extensions are designed to make sentences more concise, more readable, and more suitable as a

target language for translations from natural languages and from other CL dialects, including CLIF. None of them, however, extend the expressive power of CGIF beyond the CG core, since the semantics of every extended feature is defined by its translation to core CGIF, whose semantics is defined by its translation to the abstract syntax of Common Logic.

The following grammar rules of extended CGIF have the same definitions as the core CGIF rules of the same name: `arcSequence`, `conceptualRelation`, `negation`, `ordinaryRelation`, `text`. The following grammar rules of extended CGIF don't occur in core CGIF, or they have more options than the corresponding rules of core CGIF: `actor`, `arc`, `boolean`, `CG`, `concept`, `eitherOr`, `equivalence`, `ifThen`, `typeExpression`.

An *actor* in extended CGIF has the option of zero or more arcs following `|` instead of just one arc.

```
actor = "(", [comment], ["#?"], CGname,
        arcSequence, "|", {arc}, [endComment], ")";
```

An *arc* in extended CGIF has the options of a defining coreference label and a concept in addition to a bound coreference label.

```
arc = [comment], (reference | "*", CGname | concept);
```

A *boolean* is either a negation or a combination of negations that represent an either-or construction, an if-then construction, or an equivalence. Instead of being marked with ~, the additional negations are represented as contexts with the type labels `Either`, `Or`, `If`, `Then`, `Equiv`, `Equivalence`, or `Iff`.

```
boolean = negation | eitherOr | ifThen | equivalence;
```

A *concept* in extended CGIF permits any combination allowed in core CGIF in the same node and it adds two important options: a type field on the left side of the concept node, and a universal quantifier on the right. Four options are permitted in the type field: a type expression, a bound coreference label prefixed with `"#"`, a constant, or the empty string; a colon is required after a type expression, but optional after the other three.

```
concept = "[", [comment],
          ( (typeExpression, ":"
            | ["#?"], CGname, [":"]),
            [["@every"], "*", CGname], {reference}, CG
          | ["@every"], "*", seqmark
          ), [endComment], "]";
```

A *conceptual graph* (CG) in extended CGIF adds Boolean combinations of contexts to core CGIF.

```
CG = {concept | conceptualRelation | boolean | comment};
```

An *either-or* is a negation with a type label `Either` that contains zero or more negations with a type label `Or`.

```
eitherOr = "[", [comment], "Either", [":"],
                {"[", [comment], "Or", [":"], CG, [endComment], "]"}
                [endComment], "]";
```

An *equivalence* is a context with a type label `Equivalence` or `Equiv` that contains two contexts with a type label `Iff`. It is defined as a pair of if-then constructions, each with one of the iff-contexts as antecedent and the other as consequent.

```
equivalence = "[", [comment], ("Equivalence" | "Equiv"), [":"],
                  "[", [comment], "Iff", [":"], CG, [endComment], "]",
                  "[", [comment], "Iff", [":"], CG, [endComment], "]",
                   [endComment], "]";
```

An *if-then* is a negation with a type label `If` that contains a negation with a type label `Then`.

```
ifThen = "[", [comment], "If", [":"], CG,
                "[", [comment], "Then", [":"], CG, [endComment], "]",
                [endComment], "]";
```

A *type expression* is a lambda-expression that may be used in the type field of a concept. The symbol `@` marks a type expression, since the Greek letter λ is not available in the ASCII subset of Unicode.

```
typeExpression = "@", "*", CGname, CG;
```

Acknowledgments

The semantics of ISO standard CGIF and its presentation in this article have benefited enormously from years of discussion and collaboration with Pat Hayes and Chris Menzel.

References

Ceccato, Silvio (1961) *Linguistic Analysis and Programming for Mechanical Translation*, Gordon and Breach, New York.

Chein, Michel, ed., (1996) *Revue d'intelligence artificielle*, Numéro Spécial Graphes Conceptuels, vol. 10, no. 1.

Chen, Weidong, Michael Kifer, & David S. Warren (1993) "Hilog: A Foundation for Higher-Order Logic Programming," *Journal of Logic Programming* **15:3**, pp. 187-230.

Dau, Frithjof (2006) "Some notes on proofs with Alpha graphs," in H. Schärfe, P. Hitzler, & P. Øhrstrøm, eds., *Conceptual Structures: Inspiration and Application*, LNAI 4068, Springer, Berlin, pp. 172-188.

Davidson, Donald (1967) "The logical form of action sentences," reprinted in D. Davidson (1980) *Essays on Actions and Events*, Clarendon Press, Oxford, pp. 105-148.

Findler, Nicholas V., ed. (1979) *Associative Networks: Representation and Use of Knowledge by Computers*, Academic Press, New York.

Genesereth, Michael R., & Richard Fikes, eds. (1992) *Knowledge Interchange Format, Version 3.0 Reference Manual*, TR Logic-92-1, Computer Science Department, Stanford University.

Gentzen, Gerhard (1935) "Untersuchungen über das logische Schließen," translated as "Investigations into logical deduction" in *The Collected Papers of Gerhard Gentzen*, ed. and translated by M. E. Szabo, North-Holland Publishing Co., Amsterdam, 1969, pp. 68-131.

Hayes, Patrick (2005) "Translating semantic web languages into Common Logic," http://www.ihmc.us/users/phayes/CL/SW2SCL.html

Hayes, Patrick, & Brian McBride (2003) "RDF semantics," W3C Technical Report, http://www.w3.org/TR/rdf-mt/

Hayes, Patrick, & Chris Menzel (2001) "A semantics for the Knowledge Interchange Format," *Proc. IJCAI 2001 Workshop on the IEEE Standard Upper Ontology*, Seattle.

Hayes, Patrick, & Chris Menzel (2006) "IKL Specification Document," http://www.ihmc.us/users/phayes/IKL/SPEC/SPEC.html

Hays, David G. (1964) "Dependency theory: a formalism and some observations," *Language* **40:4**, 511-525.

Hitzler, Henrik, & Schärfe, Pascal Henrik (2009) *Conceptual Structures in Practice*, Chapman & Hall/CRC Pree, Boca Raton, FL.

ISO/IEC (2002) *Z Formal Specification Notation — Syntax, Type System, and Semantics*, IS 13568, International Organisation for Standardisation.

ISO/IEC 24707 (2007) *Common Logic (CL) — A Framework for a family of Logic-Based Languages*, International Organisation for Standardisation, Geneva, Switzerland.

Kamp, Hans (1981) "A theory of truth and semantic representation," in *Formal Methods in the Study of Language*, ed. by J. A. G. Groenendijk, T. M. V. Janssen, & M. B. J. Stokhof, Mathematical Centre Tracts, Amsterdam, 277-322.

Kamp, Hans, & Uwe Reyle (1993) *From Discourse to Logic*, Kluwer, Dordrecht.

Levinson, Robert A., & Gerard Ellis (1992) "Multilevel hierarchical retrieval," *Knowledge Based Systems* **5:3**, pp. 233-244.

Masterman, Margaret (1961) "Semantic message detection for machine translation, using an interlingua," *Proc. 1961 International Conf. on Machine Translation*, 438-475.

McDermott, Drew V. (1976) "Artificial intelligence meets natural stupidity," *SIGART Newsletter*, no. 57, April 1976.

Nagle, T. E., J. A. Nagle, L. L. Gerholz, & P. W. Eklund, eds. (1992) *Conceptual Structures: Current Research and Practice*, Ellis Horwood, New York.

Peirce, Charles Sanders (1880) "On the algebra of logic," *American Journal of Mathematics* **3**, 15-57.

Peirce, Charles Sanders (1885) "On the algebra of logic," *American Journal of Mathematics* **7**, 180-202.

Peirce, Charles Sanders (1898) *Reasoning and the Logic of Things*, The Cambridge Conferences Lectures of 1898, ed. by K. L. Ketner, Harvard University Press, Cambridge, MA, 1992.

Peirce, Charles Sanders (1906) Manuscripts on existential graphs, *Collected Papers of Charles Sanders Peirce*, vol. 4, Harvard University Press, Cambridge, MA, pp. 320-410.

Peirce, Charles Sanders (1909) Manuscript 514, with commentary by J. F. Sowa, http://www.jfsowa.com/peirce/ms514.htm

Quine, Willard Van Orman (1954) "Reduction to a dyadic predicate," *J. Symbolic Logic* **19**, reprinted in W. V. Quine, *Selected Logic Papers*, Enlarged Edition, Harvard University Press, Cambridge, MA, 1995, pp. 224-226.

Sarraf, Qusai, & Gerard Ellis (2006) "Business Rules in Retail: The Tesco.com Story," *Business Rules Journal* **7:6**, http://www.brcommunity.com/a2006/n014.html

Sowa, John F. (1976) "Conceptual graphs for a database interface," *IBM Journal of Research and Development* **20:4**, 336-357.

Sowa, John F. (1984) *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley, Reading, MA.

Sowa, John F. (2000) *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, Brooks/Cole Publishing Co., Pacific Grove, CA.

Sowa, John F. (2003) "Laws, facts, and contexts: Foundations for multimodal reasoning," in *Knowledge Contributors*, edited by V. F. Hendricks, K. F. Jørgensen, and S. A. Pedersen, Kluwer Academic Publishers, Dordrecht, pp. 145-184.

Sowa, John F. (2006) "Worlds, Models, and Descriptions," *Studia Logica*, Special Issue *Ways of Worlds II*, **84:2**, 2006, pp. 323-360.

Sowa, John F. (2010) "Peirce's tutorial on existential graphs,"

Sowa, John F., & Arun K. Majumdar (2003) "Analogical reasoning," in A. de Moor, W. Lex, & B. Ganter, eds. (2003) *Conceptual Structures for Knowledge Creation and Communication*, LNAI 2746, Springer-Verlag, Berlin, pp. 16-36.

Stewart, John (1996) *Theorem Proving Using Existential Graphs*, MS Thesis, Computer and Information Science, University of California at Santa Cruz.

Tarski, Alfred (1933) "The concept of truth in formalized languages," in A. Tarski, *Logic, Semantics, Metamathematics*, Second edition, Hackett Publishing Co., Indianapolis, pp. 152-278.

Tesnière, Lucien (1959) *Éléments de Syntaxe structurale*, Librairie C. Klincksieck, Paris. Second edition, 1965.

Way, Eileen C., ed. (1992) *Journal of Experimental and Theoretical Artificial Intelligence* (JETAI), Special Issue on Conceptual Graphs, vol. 4, no. 2.

Woods, William A. (1975) "What's in a link: foundations for semantic networks," in D. G. Bobrow & A. Collins, eds. (1975) *Representation and Understanding*, Academic Press, New York, pp. 35-82.