Relating Templates to Language and Logic

John F. Sowa

Abstract. Syntactic theories relate sentence structure to the details of morphemes, inflections, and word order. Semantic theories relate sentences to the details of formal logic and model theory. But many of the most successful programs for information extraction (IE) are based on domain-dependent templates that ignore the details at the center of attention of the major theories of syntax and semantics. This paper shows that it is possible to find a more primitive set of operations, called the *canonical formation rules*, which underlie both the template-filling operations of IE and the more formal operations of parsers and theorem provers. These rules are first stated in terms of conceptual graphs and then generalized to any knowledge representation, including predicate calculus, frames, and the IE templates. As a result, the template-filling operations of IE become part of a more general set of operations that can be used in various combinations to process knowledge of any kind, including linguistic knowledge, at any level of detail.

Presented at the Summer School on Information Extraction (SCIE99), Frascati, Italy, July 1999. Published in *Information Extraction: Towards Scalable, Adaptable Systems*, edited by M. T. Pazienza, LNAI 1714, Berlin: Springer, pp. 76-94.

1 Relating Different Language Levels

Since the 1960s, many theoretical and computational linguists have assumed that language processing, either in the human brain or in a computer program, is best performed by an integrated system of modules that operate on different language levels: phonology, morphology, syntax, semantics, pragmatics, and general world knowledge. Noam Chomsky and his students started from the bottom and defined syntactic structures for the first three levels. Richard Montague and his colleagues started in the middle with symbolic logic for the semantic level and worked downward into syntax and upward to pragmatics. Roger Schank and his students started from the top with a variety of knowledge representations: conceptual dependencies, scripts, MOPs (memory organization packets), TOPs (thematic organization packets), and problem-dependent representations for case-based reasoning about general world knowledge. Although these schools of thought shared some common views about the existence of different language levels, their theoretical foundations and notations were so radically different that collaboration among them was impossible, and adherents of different paradigms ignored each other's work.

During the 1980s, Prolog demonstrated that a general logic-based approach could be fast enough to handle every level from phonology to world knowledge. Prolog-like unification grammars for the lower levels could be integrated with deductive reasoning and possible-world semantics for the higher levels. In one form or another, logic would be the underlying mechanism that supported every level of language processing. Various notations for different subsets of logic were developed: feature structures, description logics, discourse representation structures, conceptual graphs, SNePS (semantic network processing system), and many variations of predicate calculus. Although these systems used different notations, their common basis in logic made it possible for techniques developed for any one of the systems to be adapted to most if not all of the others.

During the 1990s, however, the MUC series of message understanding conferences and the DARPA Tipster project showed that the integrated systems designed for detailed analysis at every level are too slow for information extraction. They cannot process the large volumes of text on the Internet fast enough to find and extract the information that is relevant to a particular topic. Instead, competing groups with a wide range of theoretical orientations converged on a common approach: domain-dependent templates for representing the critical patterns of concepts and a limited amount of syntactic processing to find appropriate phrases that fill slots in the templates (Hirschman & Vilain 1995).

The group at SRI International (Appelt et al. 1993; Hobbs et al. 1997) found that TACITUS, a logic-based text-understanding system was far too slow. It spent most of its time on syntactic nuances that were irrelevant to the ultimate goal. They replaced it with FASTUS, a finite-state processor that is triggered by key words, finds phrase patterns without attempting to link them into a formal parse tree, and matches the phrases to the slots in the templates. Cowrie and Lehnert (1996) observed that the FASTUS templates, which are simplified versions of a logic-based approach, are hardly distinguishable from the sketchy scripts that DeJong (1979, 1982) developed as a simplified version of a Schankian approach.

Although IE systems have achieved acceptable levels of recall and precision on their assigned tasks, there is more work to be done. The templates are hand-tailored for each domain, and their success rates on homogeneous corpora evaporate when they are applied to a wide range of documents. The high performance of template-based IE comes at the expense of a laborious task of designing specialized templates. Furthermore, that task can only be done by highly trained specialists, usually the same researchers who implemented the system that uses the templates.

A practical IE system cannot depend on the availability of human consultants for routine customization. It should automatically construct new templates from information supplied by users who have some familiarity with the domain, but no knowledge of how the IE system works. But the laborious task of deriving customized templates for a new domain is very different from the high-speed task of using the templates. Whereas the extraction task does shallow processing of large volumes of text, the customization task requires detailed understanding of the user's questions and the context in which they are asked. It depends on all the syntactic, semantic, pragmatic, and logical nuances that are ignored in the high-speed search and extraction task.

Ultimately, a practical IE system must be able to perform detailed text understanding, but on a much smaller amount of information than the search and extraction task. When deriving customized templates, the system must focus on specific information about the user's requirements. The customization task must do more than translate a single query into SQL. It may start with a query, but it must continue with a clarification dialog to resolve ambiguities and fill in background knowledge. But linking the customization stage with the extraction stage requires a common semantic framework that can accommodate both.

The purpose of this paper is to show how the IE templates fit into a larger framework that links them to the more detailed issues of parse trees, discourse structures, and formal semantics. This framework is related to logic, but not in the same way as the logic-based systems of the 1980s. Instead, it depends on a small set of lower-level operations, called the *canonical formation rules*, which were originally developed in terms of conceptual graphs (Sowa 1984). But those operations can be generalized to any knowledge representation language, including predicate calculus, frames, and IE templates. This paper presents the canonical formation rules, and relates them to conceptual graphs (CGs), predicate calculus, frames, and templates. The result is not a magic solution to all the problems, but a framework in which they can be addressed.

2 Canonical Formation Rules

All operations on conceptual graphs are based on combinations of six *canonical formation rules*, each of which performs one basic graph operation. Logically, each rule has one of three possible effects: it makes a CG more specialized, it makes a CG more generalized, or it changes the shape of a CG, but leaves it logically equivalent to the original. All the rules come in pairs: for each specialization rule, there is an inverse generalization rule; and for each equivalence rule, there is an inverse equivalence rule that transforms a CG to its original shape. These rules are fundamentally graphical: they are easier to show than to describe.

The first two rules, which are illustrated in Figure 1, are *copy* and *simplify*. At the top is a conceptual graph for the sentence *The cat Yojo is chasing a mouse*. The boxes are called *concepts*, and the circles are called *conceptual relations*. In each box is a *type label*, such as Cat, Chase, and Mouse. In the concept [Cat: Yojo], the *type field* is separated by a colon from the *referent field*, which contains the name of a specific cat named Yojo. The agent (Agnt) relation links the concept of chasing to the concept of the cat Yojo, and the theme (Thme) relation links it to the concept of a mouse.

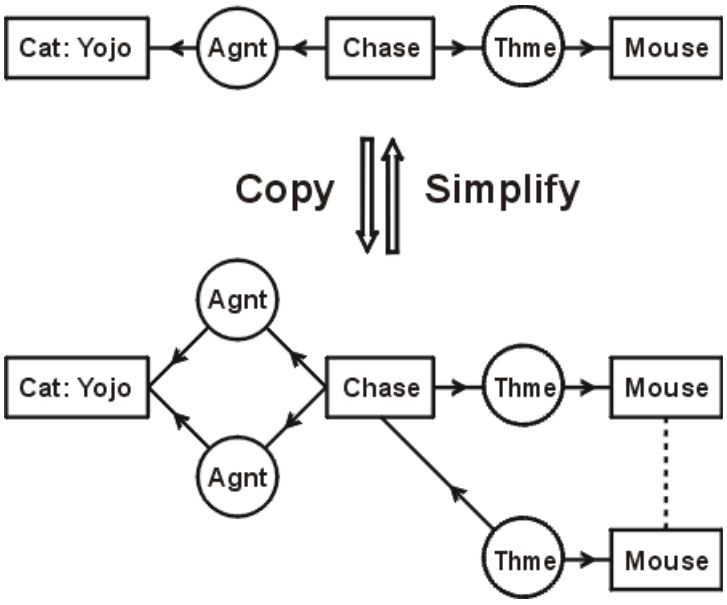

Figure 1. Copy and simplify rules

The down arrow in Figure 1 represents the copy rule. One application of the rule copies the Agnt relation, and a second application copies the subgraph →(Thme)→[Mouse]. The dotted line connecting the two [Mouse] concepts is a *coreference link* that indicates that both concepts refer to the same individual. The copies in the bottom graph are redundant, since they add no new information. The up arrow represents two applications of the simplify rule, which performs the inverse operations of erasing redundant copies. The copy and simplify rules are called *equivalence rules* because any two CGs that can be transformed from one to the other by any combination of copy and simplify rules are logically equivalent.

The two formulas in predicate calculus that are derived from the CGs in Figure 1 are also logically equivalent. In typed predicate calculus, each concept of the top CG maps to a quantified variable whose type is the same as the concept type. If no other quantifier is written in the referent field, the default quantifier is the existential ∃. The top CG maps to the following formula:

- (∃x:Cat)(∃y:Chase)(∃z:Mouse)(name(x,'Yojo') ∧agnt(y,x) ∧thme(y,z)),

which is true or false under exactly the same circumstances as the formula that corresponds to the bottom CG:

- (∃x:Cat)(∃y:Chase)(∃z:Mouse)(∃w:Mouse)(name(x,'Yojo') ∧agnt(y,x) ∧agnt(y,x) ∧thme(y,z) ∧thme(y,w) ∧z=w).

By the inference rules of predicate calculus, the redundant copy of agnt(y,x) can be erased. The equation z=w, which corresponds to the coreference link between the two [Mouse] concepts, allows the variable w to be replaced by z. After the redundant parts have been erased, the simplification of the second formula transforms it back to the first.

Figure 2 illustrates the restrict and unrestrict rules. At the top is a CG for 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*. The first step is a *restriction by referent* of the concept [Cat], which represents some indefinite cat, to the more specific concept [Cat: Yojo], which represents an individual cat named Yojo. The second step is a *restriction by type* of the concept [Animal] to a concept of the subtype [Mouse]. Two applications of the unrestrict rule perform the inverse transformation of the bottom graph to the top graph. The restrict rule is called a *specialization rule*, and the unrestrict rule is a *generalization rule*. The more specialized graph implies the more general one: if the cat Yojo is chasing a mouse, it follows that a cat is chasing an animal.

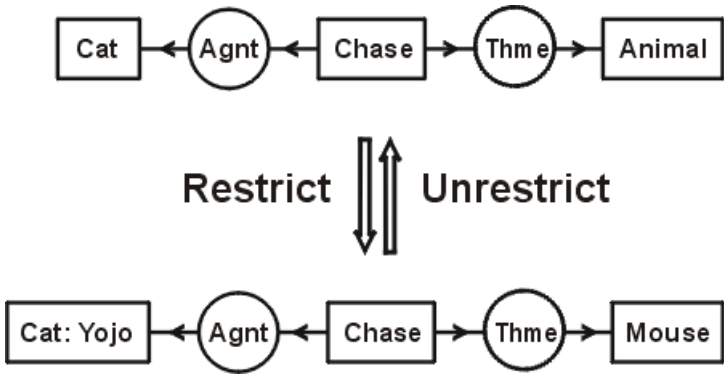

Figure 2. Restrict and unrestrict rules

Equivalent operations can be performed on the corresponding formulas in predicate calculus. The top graph corresponds to the formula

- (∃x:Cat)(∃y:Chase)(∃z:Animal)(agnt(y,x) ∧thme(y,z)).

Restriction by referent adds the predicate name(x,"Yojo"), and restriction by type replaces the type label Animal with Mouse:

- (∃x:Cat)(∃y:Chase)(∃z:Mouse)(name(x,"Yojo") ∧agnt(y,x) ∧agnt(y,x) ∧thme(y,z)).

By the rules of predicate calculus, this formula implies the previous one.

Figure 3 illustrates the *join* and *detach* rules. At the top 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 performs the inverse operation. The result of join is a more specialized graph that implies the one derived by detach.

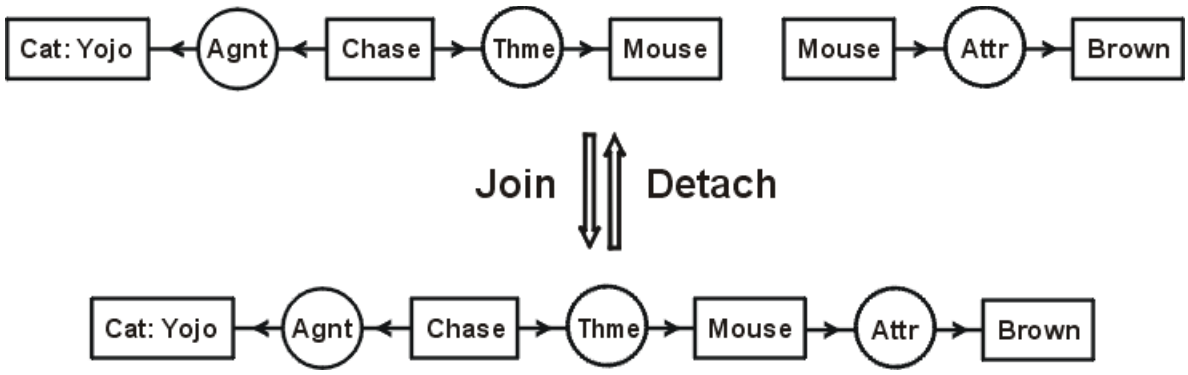

Figure 3. Join and detach rules

In predicate calculus, join corresponds to identifying two variables, either by an equality operator such as z=w or by a substitution of one variable for every occurrence of the other. The conjunction of the formulas for the top two CGs is

- ((∃x:Cat)(∃y:Chase)(∃z:Mouse)(name(x,"Yojo") ∧agnt(y,x) ∧thme(y,z)) ∧ ((∃w:Mouse)(∃v:Brown)attr(w,v)).

After substituting z for all occurrences of w and deleting redundancies,

- (∃x:Cat)(∃y:Chase)(∃z:Mouse)(∃v:Brown)(name(x,"Yojo") ∧agnt(y,x) ∧thme(y,z) ∧attr(z,w)).

By the rules of predicate calculus, this formula implies the previous one.

Although the canonical formation rules are easy to visualize, the formal specifications require more detail. They are most succinct for the *simple graphs*, which are CGs with no contexts, no negations, and no quantifiers other than existentials. The following specifications, stated in terms of the abstract syntax, can be applied to a simple graph u to derive another simple graph w.

1. *Equivalence rules*. The copy rule copies a graph or subgraph. The simplify rule performs the inverse operation of erasing a copy. Let v be any subgraph of a simple graph u; v may be empty or it may be all of u.
    - *Copy*. The copy rule makes a copy of any subgraph v of u and adds it to u to form w. If c is any concept of v that has been copied from a concept d in u, then c must be a member of exactly the same coreference sets as d. Some conceptual relations of v may be linked to concepts of u that are not in v; the copies of those conceptual relations must be linked to exactly the same concepts of u.
    - *Simplify*. The simplify rule is the inverse of copy. If two subgraphs v_1 and v_2 of u are identical, they have no common concepts or conceptual relations, and corresponding concepts of v_1 and v_2 belong to the same coreference sets, then v_2 may be erased. If any conceptual relations of v_1 are linked to concepts of u that are not in v_1, then the corresponding conceptual relations of v_2 must be linked to exactly the same concepts of u, which may not be in v_2.
2. *Specialization rules*. The restrict rule specializes the type or referent of a single concept node. The join rule merges two concept nodes to a single node. These rules transform u to a graph w that is more specialized than u.
    - *Restrict*. Any concept or conceptual relation of u may be *restricted by type* by replacing

its type with a subtype. Any concept of u with a blank referent may be *restricted by referent* by replacing the blank with some other existential referent.

- *Join*. Let c and d be any two concepts of u whose types and referents are identical. Then w is the graph obtained by deleting d, adding c to all coreference sets in which d occurred, and attaching to c all arcs of conceptual relations that had been attached to d.

3. *Generalization rules*. The unrestrict rule, which is the inverse of restrict, generalizes the type or referent of a concept node. The detach rule, which is the inverse of join, splits a graph in two parts at some concept node. The last two rules transform u to a graph w that is a generalization of u.

    - *Unrestrict*. Let c be any concept of u. Then w may be derived from u by unrestricting c either by type or by referent: unrestriction by type replaces the type label of c with some supertype; and unrestriction by referent erases an existential referent to leave a blank.
    - *Detach*. Let c be any concept of u. Then w may be derived from u by making a copy d of c, detaching one or more arcs of conceptual relations that had been attached to c, and attaching them to d.

Although the six canonical formation rules have been explicitly stated in terms of conceptual graphs, equivalent operations can be performed on any knowledge representation. The equivalents for predicate calculus were illustrated for Figures 1, 2, and 3. Equivalent operations can also be performed on frames and templates: the copy and simplify rules are similar to the CG versions; restrict corresponds to filling slots in a frame or specializing a slot to a subtype; and join corresponds to inserting a pointer that links slots in two different frames or templates.

For nested contexts, the formation rules depend on the level of nested negations. A positive context (sign +) is nested in an even number negations (possibly zero). A negative context (sign −) is nested in an odd number of negations.

- *Zero negations*. A context that has no attached negations and is not nested in any other context is defined to be positive.
- *Negated context*. The negation relation (Neg) or its abbreviation by the ~ or ¬ symbol reverses the sign of any context it is attached to: a negated context contained in a positive context is negative; a negated context contained in a negative context is positive.
- *Scoping context*. A context C with the type label SC and no attached conceptual relations is a scoping context, whose sign is the same as the sign of the context in which it is nested.

Let u be a conceptual graph in which some concept is a context whose designator is a nested conceptual graph v. The following canonical formation rules convert u to another CG w by operating on the nested graph v, while leaving everything else in u unchanged.

1. *Equivalence rules*.

    - If v is a CG in the context C, then let w be the graph obtained by performing a copy or simplify rule on v.
    - A context of type Negation whose referent is another context of type Negation is called a *double negation*. If u is a double negation around that includes the graph v, then let w be the graph obtained by replacing u with a scoping context around v:

      ```
      [Negation: [Negation: v]]  =>  [SC: v].
      ```

A double negation or a scoping context around a conceptual graph may be drawn or erased at any time. If v is a conceptual graph, the following three forms are equivalent:

```
~[ ~[ v]],  [v],  v.
```

2. *Specialization rules*.
    - If C is positive, then let w be the result of performing any specialization rule in C.
    - If C is negative, then let w be the result of performing any generalization rule in C.
3. *Generalization rules*.
    - If C is positive, then let w be the result of performing any generalization rule in C.
    - If C is negative, then let w be the result of performing any specialization rule in C.

In summary, negation reverses the effect of generalization and specialization, but it has no effect on the equivalence rules. Corresponding operations can be performed on formulas in predicate calculus. For frames and templates, the treatment of negation varies from one implementation to another; some systems have no negations, and others have many special cases that must be treated individually. But for any knowledge representation that supports negation, the same principle holds: negation reverses generalization and specialization.

3 Notation-Independent Rules of Inference

The canonical formation rules, which can be formulated in equivalent versions for conceptual graphs and predicate calculus, extract the logical essence from the details of syntax. As a result, the rules of inference can be stated in a notation-independent way. In fact, they are so completely independent of notation that they apply equally well to any knowledge representation for which it is possible to define rules of generalization, specialization, and equivalence. That includes frames, templates, discourse representation structures, feature structures, description logics, expert system rules, SQL queries, and any semantic representation for which the following three kinds of rules can be formulated:

- *Equivalence rules*. The equivalence rules may change the appearance of a knowledge representation, but they do not change its logical status. If a graph or formula u is transformed to another graph or formula v by any equivalence rule, then u implies v, and v implies u.
- *Specialization rules*. The specialization rules transform a graph or formula u to a graph or formula v that is logically more specialized: v implies u.
- *Generalization rules*. The generalization rules transform a graph or formula u to a graph or formula v that is logically more generalized: u implies v.

The notation-independent rules of inference were formulated by the logician Charles Sanders Peirce. Peirce (1885) had originally invented the algebraic notation for predicate calculus with notation-dependent rules for *modus ponens* and instantiation of universally quantified variables. But he continued to search for a simpler and more general representation, which expressed the logical operations diagrammatically, in what he called a more *iconic* form. In 1897, Peirce invented *existential graphs* and introduced rules of inference that depend only on the operations of copying, erasing, and combining graphs. These five rules are so general that they apply to any version of logic for which the corresponding operations can be defined:

1. *Erasure*. In a positive context, any graph or formula u may be replaced by a generalization of u; in particular, u may be erased (i.e. it may replaced by a blank, which is the universal

generalization).

2. *Insertion*. In a negative context, any graph or formula u may be replaced by a specialization of u; in particular, any graph may be inserted (i.e. it may replace the blank).
3. *Iteration*. If a graph or formula u occurs in a context C, another copy of u may be drawn in the same context C or in any context nested in C.
4. *Deiteration*. Any graph or formula u that could have been derived by iteration may be erased.
5. *Equivalence*. Any equivalence rule (copy, simplify, or double negation) may be performed on any graph, subgraph, formula, or subformula in any context.

Each of these rules preserves truth: if the starting graph or formula u happens to be true, the resulting formula v must also be true. Peirce's only axiom is the blank *sheet of assertion*. A blank sheet, which says nothing, cannot be false. Any statement that is derivable from the blank by these rules is a *theorem*, which must always be true.

When applied to entire graphs or formulas, these rules support propositional logic; but when they are applied to subgraphs and coreference links, they support full first-order logic. Peirce's rules take their simplest form when they are applied to his original existential graphs or to conceptual graphs, which are a typed version of existential graphs. When they are applied to the predicate calculus notation, Peirce's rules must accommodate various special cases that depend on the properties of each of the logical operators. That accommodation transforms Peirce's five rules to the rules of *natural deduction*, which were defined by Gerhard Gentzen over thirty years later. For Peirce's original statement of the rules, see Roberts (1973). For further examples and discussion of their application to other logics, see Sowa (1998, 2000).

4 Generalization Hierarchies

The rules of inference of logic define a generalization hierarchy over the terms of any logic-based language. Figure 4 shows a hierarchy in conceptual graphs, but an equivalent hierarchy could be represented in any knowledge representation language. For each dark arrow in Figure 4, the graph above is a generalization, and the graph below is a specialization. The top graph says that an animate being is the agent (Agnt) of some act that has an entity as the theme (Thme) of the act. Below it are two specializations: a graph for a robot washing a truck, and a graph for an animal chasing an entity. Both of these graphs were derived from the top graph by repeated applications of the rule for restricting type labels to subtypes. The graph for an animal chasing an entity has three specialization: a human chasing a human, a cat chasing a mouse, and the dog Macula chasing a Chevrolet. These three graphs were also derived by repeated application of the rule of restriction. The derivation from [Animal] to [Dog: Macula] required both a restriction by type from Animal to Dog and a restriction by referent from the blank to the name Macula.

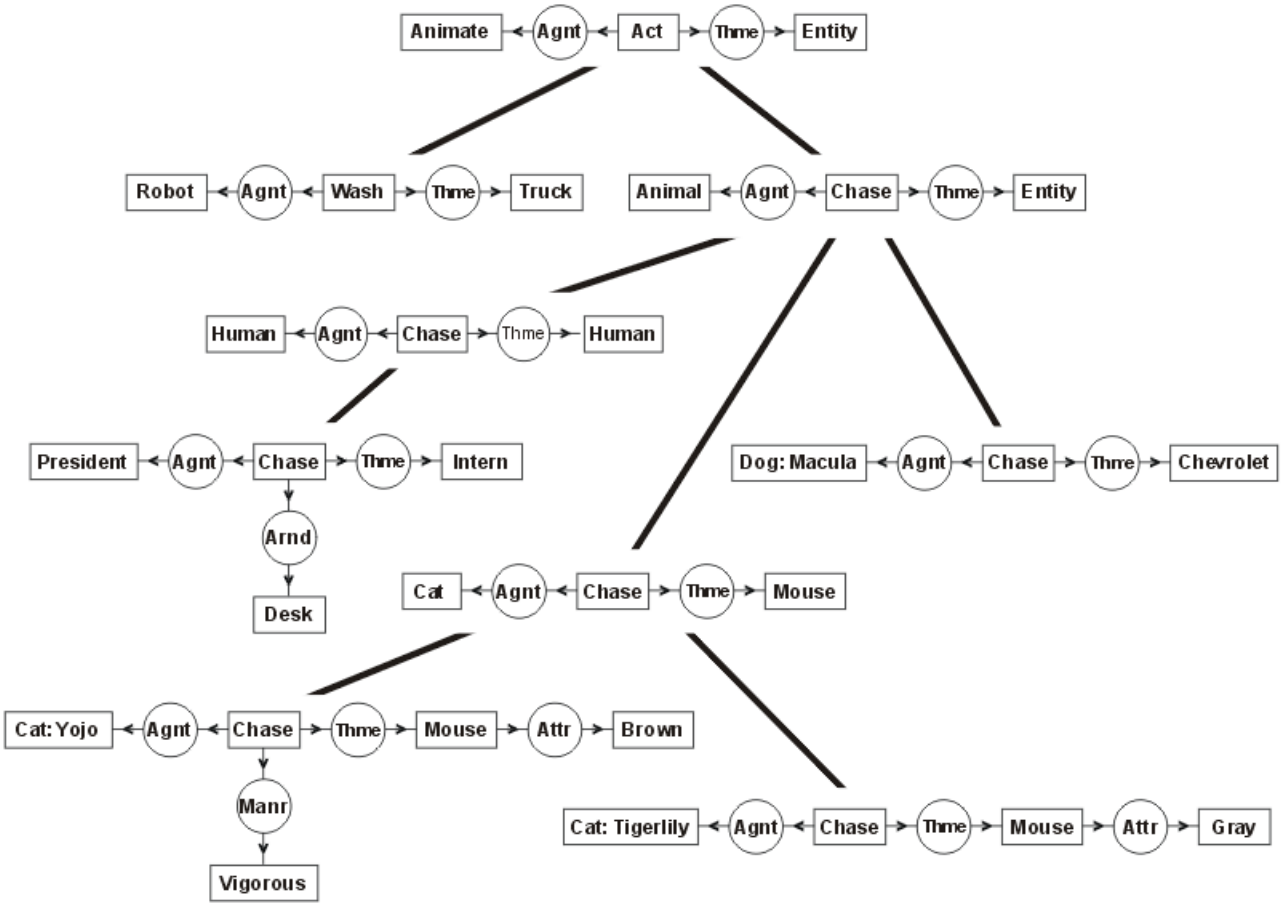

Figure 4. A generalization hierarchy

Besides restriction, a join was used to specialize the graph for a human chasing a human to the graph for a senator chasing a secretary around a desk. The join was performed by merging the concept [Chase] in the upper graph with the concept [Chase] in the following graph:

```
[Chase]→(Arnd)→[Desk].
```

Since the resulting graph has three relations attached to the concept [Chase], it is not possible to represent the graph on a single line in a linear notation. Instead, a hyphen may be placed after the concept [Chase] to show that the attached relations are continued on subsequent lines:

```
[Chase]-
   (Agnt)→[Senator]
   (Thme)→[Secretary]
   (Arnd)→[Desk].
```

For the continued relations, it is not necessary to show both arcs, since the direction of one arrow implies the direction of the other one.

The two graphs at the bottom of Figure 4 were derived by both restriction and join. The graph on the left says that the cat Yojo is vigorously chasing a brown mouse. It was derived by restricting [Cat] to [Cat: Yojo] and by joining the following two graphs:

```
[Mouse]→(Attr)→[Brown].
[Chase]→(Manr)→[Vigorous].
```

The relation (Manr) represents manner, and the relation (Attr) represents attribute. The bottom right graph of Figure 4 says that the cat Tigerlily is chasing a gray mouse. It was derived from the graph

above it by one restriction and one join. All the derivations in Figure 4 can be reversed by applying the generalization rules from the bottom up instead of the specialization rules from the top down: every restriction can be reversed by unrestriction, and every join can be reversed by detach.

The generalization hierarchy, which is drawn as a tree in Figure 4, is an excerpt from a lattice that defines all the possible generalizations and specializations that are possible with the rules of inference. Ellis, Levinson, and Robinson (1994) implemented such lattices with high-speed search mechanisms for storing and retrieving graphs. They extended their techniques to systems that can access millions of graphs in time proportional to the logarithm of the size of the hierarchy. Their techniques, which were designed for conceptual graphs, can be applied to any notation, including frames and templates.

5 Frames and Templates

Predicate calculus and conceptual graphs are domain-independent notations that can be applied to any subject whatever. In that respect, they resemble natural languages, which can express anything that anyone can express in any artificial language plus a great deal more. The templates for information extraction and the frames for expert systems are usually highly specialized to a particular application. One expert system for diagnosing cancer patients represented knowledge in a frame with the following format:

```
(defineType MedPatient
  (supertype Person)
  ...
  (motherMelanoma (type Boolean)
                  (question '(Has the patient's mother had melanoma?)) ))
```

This frame says that a medical patient, MedPatient, has a supertype Person. Then it lists several attributes, including one named motherMelanoma, which has two *facets*: one facet declares that the values of the attribute must be of type Boolean; and the other specifies a character string called a question. Whenever the system needs the current value of the motherMelanoma attribute, it prints the character string on the display screen, and a person answers yes or no. Then the system converts the answer to a Boolean value (T or F), which becomes the value of the attribute.

Such frames are simple, but they omit important details. The words *mother* and *melanoma* appear in a character string that is printed as a question for some person at a computer display. Although the person may know the meaning of those words, the system cannot relate them to the attribute motherMelanoma, which by itself has no more meaning than the character string "MM". Whether or not the system can generate correct answers using values of that attribute depends on how the associated programs happen to process the character strings.

To express those details, Figure 5 shows a conceptual graph for the sentence *The patient's mother suffered from melanoma*.

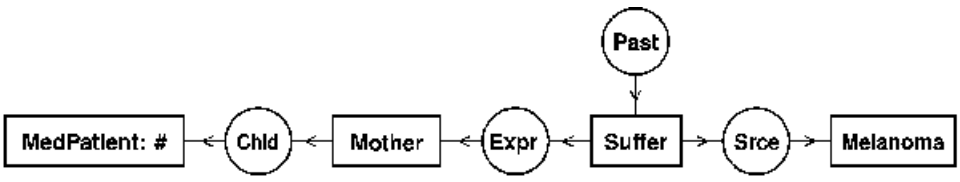

Figure 5. CG for the English sentence

The concept [MedPatient: #] for the medical patient is linked via the child relation (Chld) to the concept [Mother]. The experiencer (Expr) and source (Srce) relations link the concept [Suffer] to the concepts [Mother] and [Melanoma]. The type hierarchy would show that a medical patient is a type of

person; a mother is a woman, which is also a person; and melanoma is a type of cancer, which is also a disease:

```
MedPatient < Person
Mother < Woman < Person
Melanoma < Cancer < Disease
```

The type MedPatient could be introduced by the following definition:

```
type MedPatient(*x) is
   [Person: ?x]←(Ptnt)←[TreatMed]→(Agnt)→[Physician].
```

In this graph, the verb *treat* is represented by the type TreatMed to distinguish it from other word senses, such as giving the patient some candy. The definition may be read *The type MedPatient is defined as a person x who is treated by a physician*.

Figure 6 defines MotherMelanoma as a dyadic relation that links a person x to a Boolean value y, where y is the mode of a proposition that x is the child of a mother who suffered from Melanoma.

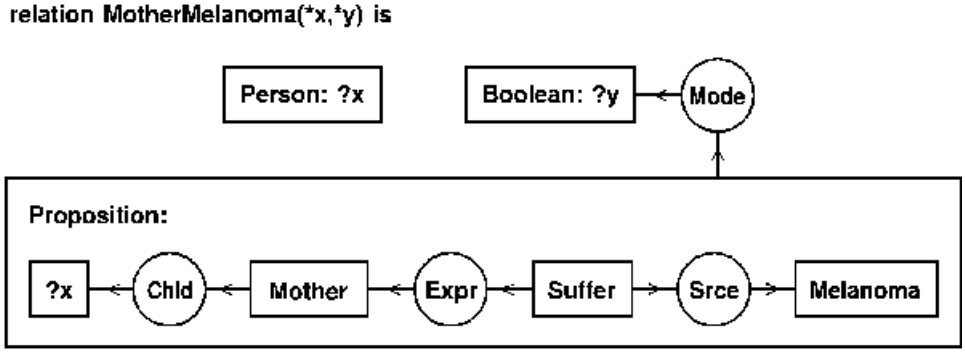

Figure 6. Definition of the MotherMelanoma relation

The amount of detail required for semantics shows why most expert systems do not have a natural language interface. A system that translated *The patient's mother had melanoma* to Figure 5 would be more complex than a typical expert system shell. Even then, it would still have to perform further inferences before it could deduce T or F for the motherMelanoma attribute. It is easier to print out a character string and let the user select yes or no from a menu. Yet that solution might make it harder to modify, extend, and link the system to other programs designed for other purposes. Consider some possible extensions:

- If the expert system were later linked to a database so that it could check the patient's records directly, the attributes stored in the database would probably not match the ones needed by the expert system. Instead of storing a motherMelanoma attribute for each patient, the database would be more likely to list the person's mother. Before the system could determine the value of that attribute, someone would have to write a database query or view with the more detailed concepts of Mother, Disease, and Melanoma.
- Besides the prompts, many expert systems include additional character strings for explanations. Many of them are stored as canned phrases that are typed whenever a particular rule is executed. But better explanations would require some tailoring of the strings to insert context-dependent values. As the system becomes more complex, the help, prompts, and explanations could take more space than the rules, and they would have to be revised or rewritten whenever the rules are changed.

- Writing the help, prompt, and explanation messages and keeping them consistent with every modification to the rules is difficult enough with one language, but for a system used in different countries, they would all have to be written and revised for every language.

These issues, which were illustrated in terms of frames, apply equally well to the frame-like IE templates. The definitional mechanisms enable the specialized frames or templates to be defined in terms of the more general representations that support the full detail of the statements in natural language.

6 Lambda Expressions

Besides the basic operators and quantifiers, a system of logic requires some method for defining new types and relations. The traditional method, illustrated in Section 5, is to use a variable, called a *formal parameter*, that appears in on both sides of the definition:

```
type MedPatient(*x) is
   [Person: ?x]←(Ptnt)←[TreatMed]→(Agnt)→[Physician].
```

In this example, the left side of the keyword **is** specifies the name MedPatient, and the right side is an expression that defines it. The symbol x, which links the two sides, is the formal parameter. Such definitions are convenient for defining named types and relations, but the connection of the definition to the name is an unnecessary complication. It creates difficulties for the operations of logic and computation, which may generate intermediate expressions that have no names.

As a systematic method for defining and evaluating functions and relations, the logician Alonzo Church (1941) invented the *lambda calculus*. In Church's notation, the name can be written by itself on one side of an equation; on the other side, the Greek letter λ is used to mark the formal parameter. That technique, which Church defined for predicate calculus, applies equally well to conceptual graphs:

```
MedPatient  =
   [Person: λ]←(Ptnt)←[TreatMed]→(Agnt)→[Physician].
```

With this notation, the name is completely separated from the defining expression, which can be used in any position where the name is used. In particular, a lambda expression can replace the type label of a concept like [MedPatient] or the type constraint on a variable in typed predicate calculus.

To support the lambda notation, Church defined rules of *lambda conversion* for expanding and contracting the expressions. One of the central results of the lambda calculus is the *Church-Rosser theorem*, which says that a nest of multiple lambda expressions may be expanded or contracted in any order and the results will always be equivalent. In effect, the lambda calculus treats functions and relations as *first-class data types*, which can be manipulated and processed like any other kind of data. The equivalent of Church's rules can be stated for conceptual graphs and many other knowledge representations:

- *Type expansion*. Let g be a conceptual graph containing a concept c that has a defined type t and an existential referent (i.e. a blank or ∃as its quantifier). Then the operation of *type expansion* performs the following transformations on g:
    1. The definition of t must be a monadic lambda expression whose body is a conceptual graph b and whose formal parameter p is a concept of b. Let the signature of t be the sequence (s), where s is the type of p.
    2. Remove the type t from the concept c of g, and replace it with s.
    3. Remove the λ marker from the concept p, and replace it with the designator of c.

4. At this point, the concepts c and p should be identical. Combine the graphs g and b by joining p to c.

- *Relational expansion*. Let g be a conceptual graph containing an n-adic relation r that has a defined type t. Then the operation of *relational expansion* performs the following transformations on g:
    1. The definition of t must be an n-adic lambda expression whose body is a conceptual graph b and whose formal parameters $<p_1,...,p_n>$ are concepts of b. Let the signature of t be the sequence $(s_1,...,s_n)$, where each s_i is the type of p_i.
    2. Remove the relation r from g, and detach each arc i of r from the concept c_i of g.
    3. Remove the λ marker from each parameter p_i, and replace it with the designator of c_i.
    4. For each i from 1 to n, restrict the types of c_i and p_i to their maximal common subtypes. Then combine the graphs g and b by joining each p_i to c_i.

The corresponding rules of *lambda contraction* are the inverses of the rules of lambda expansion: if any conceptual graph g could have been derived by lambda expansion from a CG u, then g may be *contracted* to form u. The expansion rules and the contraction rules are equivalence rules, which change the appearance of a graph or formula, but they do not change its truth or falsity. Those rules can be used as equivalence operations in the rules of inference stated in Section 3.

7 The IE Paradigm

As many people have observed, the pressures of extracting information at high speed from large volumes of text have led to a new paradigm for computational linguistics. Appelt et al. (1993) summarized the IE paradigm in three bullet points:

- Only a fraction of the text is relevant; in the case of the MUC-4 terrorist reports, probably only about 10% is relevant.
- Information is mapped into a predefined, relatively simple, rigid target representation; this condition holds whenever entry of information into a database is the task.
- The subtle nuances of meaning and the writer's goals in writing the text are of no interest.

They contrast the IE paradigm with the more traditional task of text understanding:

- The aim is to make sense of the entire text.
- The target representation must accommodate the full complexities of language.
- One wants to recognize the nuances of meaning and the writer's goals.

At a high level of abstraction, this characterization by the logicians at SRI International would apply equally well to all the successful competitors in the MUC and Tipster evaluations. Despite the differences in their approaches to full-text understanding, they converged on a common approach to the IE task. As a result, some observers have come to the conclusion that IE is emerging as a new subfield in computational linguistics.

The convergence of different approaches on a common paradigm is not an accident. At both ends of the research spectrum, the logicians and the Schankians believe that the IE paradigm is a special case of their own approach, despite their sharp disagreements about the best way to approach the task of full-text understanding. To a certain extent, both sides are right, because both the logic-based approach

and the Schankian approach are based on the same set of underlying primitives. The canonical formation rules of generalization, specialization, and equivalence can be used to characterize all three approaches to language processing that were discussed in Section 1:

- *Chomskyan*. The starting symbol S for a context-free grammar is a universal generalization: every sentence that is derivable from S by a context-free grammar is a specialization of S, and the parse tree for a sentence is a record of the sequence of specialization rules used to derive it from S. Chomsky's original goal for transformational grammar was to define the equivalence rules that preserve meaning while changing the shape or appearance of a sentence. The evolution of Chomsky's theories through the stages of government and binding (GB) theory to his more recent minimalism has been a search for the fundamental equivalence rules of Universal Grammar.
- *Montagovian*. Instead of focusing on syntax, Montague treated natural language as a disguised version of predicate calculus. His categorial grammars for deriving a sentence are specialization rules associated with lambda-expressions for deriving natural language sentences. Hobbs et al. (1993) explicitly characterized the semantic interpretation of a sentence as abduction: the search for a specialized formula in logic that implies the more generalized subformulas from which it was derived.
- *Schankian*. Although Roger Schank has denounced logic and logicians as irrelevant, every one of his knowledge representations can be defined as a particular subset of logic with an idiosyncratic notation. Most of them, in fact, represent the existential-conjunctive (EC) subset of logic, whose only operators are the existential quantifier and conjunction. Those two operators, which happen to be the most frequently used operators in formulas derived from natural language text, are also the two principal operators in discourse representation theory, conceptual graphs, and Peirce's existential graphs. The major difference is that Schank has either ignored the other operators or treated them in an ad hoc way, while the logicians have generalized their representations to accommodate all the operators in a systematic framework.

In summary, the canonical formation rules reveal a level of processing that underlies all these approaches. The IE templates represent the special case of EC logic that is common to all of them. The detailed parsing used in text understanding and the sketchy parsing used in IE are both applications of specialization rules; the major difference is that IE focuses only on that part of the available information that is necessary to answer the immediate goal. The subset of information represented in the IE templates can be derived by lambda abstractions from the full information, as discussed in Sections 5 and 6. This view does not solve all the problems of the competing paradigms, but it shows how they are related and how innovations in one approach can be translated to equivalent techniques in the others.

References

Appelt, Douglas E., Jerry R. Hobbs, John Bear, David Israel, & Mabry Tyson (1993) FASTUS: A finite-state processor for information-extraction from real-world text, in *Proceedings of IJCAI 93*, pp. 1172-1178.

Church, Alonzo (1941) *The Calculi of Lambda Conversion*, Princeton: University Press.

Cowie, Jim, & Wendy Lehnert (1996) Information extraction, *Communications of the ACM* **39:1**, 80-91.

DeJong, Gerald F. (1979) Prediction and substantiation, *Cognitive Science* **3**, 251-273.

DeJong, Gerald F. (1982) An overview of the FRUMP system, in W. G. Lehnert & M. H. Ringle, eds., *Strategies for Natural Language Processing*, Hillsdale, NJ: Erlbaum, pp. 149-176.

Ellis, Gerard, Robert A. Levinson, & Peter J. Robinson (1994) Managing complex objects in Peirce, *International J. of Human-Computer Studies* **41**,. 109-148.

Hirschman, Lynette, & Marc Vilain (1995) Extracting Information from the MUC, ACL Tutorial, Cambridge, MA: MIT.

Hobbs, Jerry R., Mark Stickel, Douglas Appelt, & Paul Martin (1993) Interpretation as abduction, *Artificial Intelligence* **63:1-2**, 69-142.

Hobbs, Jerry R., Douglas Appelt, John Bear, David Israel, Megumi Kameyama, Mark Stickel, & Mabry Tyson (1997) FASTUS: A cascaded finite-state transducer for extracting information from natural-language text, in E. Roche & Y. Schabes, eds., *Finite-State Language Processing*, Cambridge, MA: MIT Press, pp. 383-406.

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

Roberts, Don D. (1973) *The Existential Graphs of Charles S. Peirce*, The Hague: Mouton.

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

Sowa, John F. (1998) The infinite variety of logics, in O. Herzog & A. Günter, eds., *KI-98: Advances in Artificial Intelligence*, LNAI 1504, Berlin: Springer-Verlag, pp. 31-53.

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