Additions for SET Representation and Processing to Conceptual Programming

Heather D. Pfeiffer and Roger T. Hartley

Knowledge Systems Group
Computing Research Laboratory
New Mexico State University
Box 30001/3CRL
Las Cruces, New Mexico 88003
Phone: 1-505-646-4143
email:hdp@nmsu.edu and rth@nmsu.edu

Abstract

New **SET** additions have been incorporated into the Conceptual Programming environment, CP, and the Model Generative Reasoning architecture, MGR, being developed here at the Computing Research Laboratory (CRL). These additions have been made at both a representational and operational level within the CP system and at a control level within the MGR system. Sets are viewed as an extension to individualization with a single individual, and also as an downward extension to the type hierarchy. The revisions to the CP and MGR systems handle these issues without departing from basic conceptual graph theory.

1 Introduction

The Conceptual Programming environment, CP, is an ongoing project here at CRL, and is based on developing a knowledge representation environment with a graphical foundation (Hartley, 1986, Pfeiffer, 1986, and Hartley and Coombs, 1989). These graphs are based on Sowa's conceptual structures (Sowa, 1984) and have defined operations on those structures. However, CP also has extended features for specifying both syntactic and semantic aspects of the representation. In the past two years, Eshner and Hartley, (1988), and Pfeiffer and Hartley, (1989), have presented semantic additions to the CP environment. This year we will define the syntax of the representation more formally and give the syntax for representing SETs within the CP system.

One application built on top of the CP environment is a problem solving system. The process of solving a problem is performed by constructing a graph, called a *model*, out of graphs that can be thought of as data, definitions and previously created models. The graph operations *join* and *project* are two of the operations performed in building the actual models. Because solutions are found by generating models during the reasoning process, the general approach has been termed "Model Generative Reasoning, MGR".

Part of the definition of conceptual structures includes concepts organized in a type lattice (Sowa, 1984). When structures built out of these concepts are operated upon with join and project, it often arises that the same concept can be specialized with different individuals. In many situations it is preferable to keep them separated, but there are cases where further processing should be done by assembling the individuals into a set. This operational need is in addition to the more obvious one of representing propositions about sets of objects in the conceptual structure language (Gardiner et al, 1989). This paper is more aimed at the former need than the latter.

2 Representation

The syntactic representation for sets within CP has been broken down into two areas: 1) types of sets and their interrelationship, and 2) the actual parsable labels of sets. The types of sets arise from the need to organize together basic concepts that have more than one individualization. The actual parsable labels give a feel for the actual syntax that the CP environment provides.

2.1 Types of Sets

The following types of sets been formulated within the CP system:

Basic sets:

- Open Sets - sets delimited by using "{}" (for 'Unordered' sets) or "()" (for 'Ordered' sets); these sets are potentially infinite and possibly have a variable length. Internal to the CP and MGR systems, they are referred to as "?SET"[1].
- Atomic Sets - sets delimited by "[]" (for 'Unordered' sets) or "⟨⟩" (for 'Ordered' sets) as a notation; these sets operate as a 'unit' and have a fixed length. They are operated on as if they are a single individual that specializes a concept, i.e. the set cannot be broken up or added to. Internal to the CP and MGR systems, they are referred to as "?GROUP".

As mentioned above, within these basic sets, there are two sub-types of set:

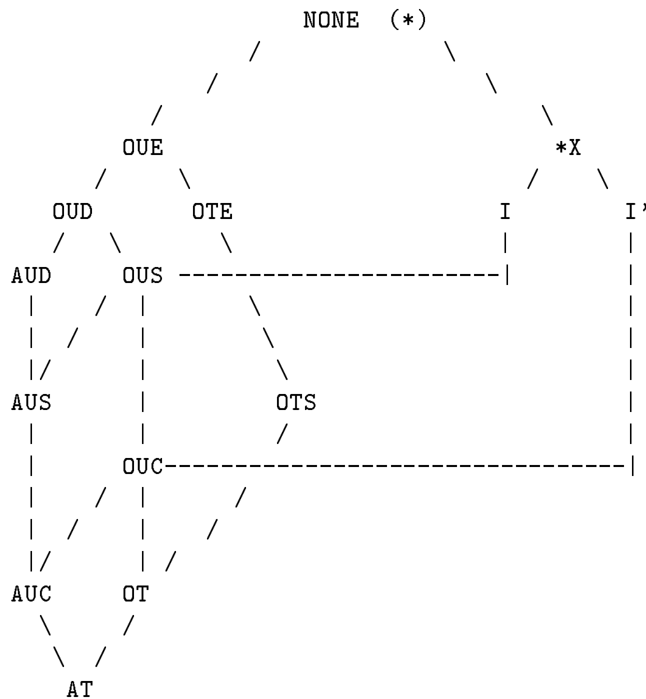
- Ordered - members are separated by ','; they display an ordering of elements. Internal to the systems, they are referred to as "O*".
- Unordered - members are separated by ',' or ';'; they will have no particular order of elements.

Within 'Unordered' sets, there are three possibilities:

- Conjunctive ("and") – Sowa's 'collective' set, where members are separated by ','; each member of the set is "anded" with the other members of the set. Internal to the systems, they are referred to as "C*".
- Disjunctive ("or") – Sowa's 'distributive' set, where members are separated by ';'; each member is 'ored' with the other members of the set. Internal to the systems, they will be referred to as "D*".
- Single Item – the set only has one member. This only applies to "Open" Sets represented in "Open" form; they are assumed to be conjunctive in nature when ambiguities arise. Internal to the systems, they will be referred to as "S*".

These types of sets allow the representation of concepts that may have more than one individual, *Open Sets*, and concepts in which a single individual is a "set of things", *Atomic Sets*. As in mathematics, these sets may have a membership of elements with no implied order, *Unordered*, or a list of elements in which the order is essential, *Ordered*.
These types of sets have a number of possible interrelationships to each other. In fact, a generalization lattice can be formed, and is displayed below. The idea is that each of the attribute pairs have a generalization relation. Thus, an ordered set (a *tuple*) is less general than an unordered one, as long as the other attributes remaining the same, i.e. $T < U$. Similarly, atomic sets with fixed size are less general than open sets which are potentially infinite, i.e. $A < O$. Finally a conjunctive set is less general than a disjunctive set, i.e. $C < D$. The empty set, denoted by the property E is more general than any, and the singleton set, denoted by the attribute S is more general than a multi-element set (adding members specializes a set). We have also included the simple individual as I in the lattice, and a further manifestation of the simple individual as I' for reasons that will become clear when we discuss set processing. The top node of the lattice is the absence of an individual.

[1] In this and subsequent notations, the '?' is a place holder for a single letter, and '*' for any number of letters




```
Key: X = variable
     I, I' = simple indivual
     E = empty
     S = singleton
     O = open
     A = atomic
     U = unordered
     T = tuple
     C = conjunctive
     D = disjunctive
```

2.2 Actual Label Display

Below are given the actual parsable labels used within the syntax of sets within the CP system:[2]

Possible “Open” sets: Infinite size:

{*}	Infinite Unordered Set (assumed to be conjunctive)
(*)	Infinite Ordered Set (assumed to be conjunctive)
{A}	Infinite Unordered Singleton Set (assumed to be conjunctive)
(A)	Infinite Ordered Set (assumed that A will always be the first element in the set, and to be conjunctive)
{A,B,C}	Infinite Unordered Conjunctive Set
(A,B,C)	Infinite Ordered Conjunctive Set
{A;B;C}	Infinite Unordered Disjunctive Set

[2] Of course members of sets may be sets themselves, but here we just give sets of names for simplicity

Size parameter:

{*}<2	Size of Unordered Set must not be greater than 1
(*)<2	Size of Ordered Set must not be greater than 1
{A,B,C}>2	Size of Unordered Set must be greater than 2
(A,B,C)>2	Size of Ordered Set must be greater than 2
{A,B,C}<5	Size of Unordered Set must not be greater than 4
(A,B,C)<5	Size of Ordered Set must not be greater than 4

NOTE: no size parameter on a disjunctive "open" set; also, size parameters have not been implemented yet.

Possible "Atomic" sets:

[A]	Finite Unordered Set
$\langle A \rangle$	Finite Ordered Set
[A,B]	Finite Unordered Conjunctive Set
$\langle A, B \rangle$	Finite Ordered Conjunctive Set
[C;B]	Finite Unordered Disjunctive Set

NOTE: no size parameter needed because no members can be added or subtracted from these sets; also, the '*' element does not exist because these sets are finite, never empty.

2.3 Sets as data structures

It may be useful to identify the various set types as data structures whose names and operational properties are more familiar. Starting with an unordered open set, we have the usual 'mathematical' *set*, that is almost never a basic data structure, but is implemented as an ADT. If we fix the size of the set (making it atomic), but keep it unordered we get the idea of a *group*, but as a data structure this would be a *record* or *structure*. An ordered open set is a *list*, and adding the property of atomicity turns the list into a *tuple*. Finally there is no easy correspondence to the conjunctive/disjunctive property, but the notion of an *enumeration* captures some of the disjunctive flavor.
This range of set types, i.e *set, group, list* and *tuple* allow the operational side of conceptual structures to be expanded enormously by giving them the flavor of data structure (more than in most programming languages) while retaining the basic philosophy.

3 Processing

Using the representation given above, the CP operators *join* and *project* can manipulate sets through a group of functions that follow the rules laid out by the generalization lattice.
Below is an example of some of these functions and what they do:

- Set-Typep: this procedure returns if a particular individual's type is one of the possible SET types. Set-Members: this procedure will parse the "set" individual within a node, producing its internal form.
- Make-Set-Ind: this procedure will read the internal form, and produce an individual label for a set.
- Set-Ind-CompatibleP: this procedure returns if a set and an individual are compatible.
- Set-Ind-Compatible: this procedure returns the spec, gen, individual value of the spec, and the individual value of the gen for a set and individual.
- Set-CompatibleP: this procedure returns if two sets are compatible.
- Set-Compatible: this procedure returns the spec, gen, individual value of the spec, and the individual value of the gen for two sets; i.e. the two sets are "joined" or "projected" if they are: 1) of the same type, 2) of the same ordering class, and 3) of the same organization.

- Set-Fix: this procedure will toggle between the two basic types of sets. Given an open set (as input), it will return an atomic and visa versa.
- Fix-Hier: this procedure will call the “Set-Fix” procedure, but will allow the users to work between types in the type hierarchy.

3.1 Join

Sowa's join operation is defined on concept types only. It allows two graphs to be merged on a concept node where the two concept types have a maximum common subtype that is not $\perp$, the absurd type. We can extend this to cover individuals if we treat the set/individual lattice in the same way as the concept type lattice is. Thus, for example, an atomic, unordered disjunctive (AUD) set such as [A;B;C] will join to an open, unordered conjunctive (OUC) set such as {B,C,D} to give an atomic, unordered conjunctive (AUC) set [A,B,C,D].
This works because of two things. Firstly, AUC is the maximum common subtype of AUD and OUC, according to the lattice. Secondly, the membership of the resultant set is the union of the memberships of the original two sets. This latter constraint follows the nature of join as an *abductive* operation (Hartley, 1990). For concept types A, B and C, where C is a subtype of A and B, the join of A and B produces C because

$$(C \rightarrow A) \wedge (C \rightarrow B)$$

from the type lattice. Similarly from the set lattice,

$$(AUC \rightarrow AUD) \wedge (AUC \rightarrow OUC)$$

hence the join of AUD and OUC produces AUC.
We can now explain the presence of two simple individuals I and I' in the lattice. The join of two unequal, simple individuals is now a conjunctive set of both of them[3]. Thus the join of (names) A and B is {A,B}. Again the logical implications work correctly:

$$(\{A,B\} \rightarrow A) \wedge (\{A,B\} \rightarrow B)$$

Tables 1 and 2 are a reasonably complete list of examples showing how the join operation works with the various types of sets and with simple individuals.
The following is a transcript of a suite of tests of the actual software routines that process set joins. It contains more detailed examples of the above canonical forms, and also examples of joining sets of sets together.

```
Individual with itself

"a" and "a" are compatible
"@345" and "@345" are compatible
"#123" and "#123" are compatible
"^state" and "^state" are compatible
"aname" and "aname" are compatible
"@456" and "@456" are compatible
"#345" and "#345" are compatible
"^state2" and "^state2" are compatible
"{*}" and "{*}" are compatible
"{a}" and "{a}" are compatible
"{c}" and "{c}" are compatible
"{e}" and "{e}" are compatible
"{a,b,c}" and "{a,b,c}" are compatible
"{c,d,e}" and "{c,d,e}" are compatible
```

[3]We realize that some practitioners will not like this, but it is not hard to think of examples from reasoning where it is essential

Table 1: Joins with Open sets

I. *SET with individuals

{*}	does join	A	=>	{A}
(*)	does join	A	=>	(A)
{A}	does join	A	=>	{A}
(A)	does join	A	=>	(A)
{A,B,C}	does join	C	=>	{A,B,C}
{A;B;C}	does join	C	=>	{C}
(A,B)	does join	A	=>	(A,B)
{A,B,C}	does join	E	=>	{A,B,C,E}
{A;B;C}	does join	D	=>	{*}
(A,C)	does join	B	=>	(A,C,B)

II. *SET with single element sets

{*}	does join	{A}	=>	{A}
(*)	does join	{A}	=>	(A)
{A}	does join	{A}	=>	{A}
(A)	does join	{A}	=>	(A)
{A,B,C}	does join	{C}	=>	{A,B,C}
{A,B,C}	does join	{E}	=>	{A,B,C,E}
{A;B;C}	does join	{C}	=>	{C}
{A;B;C}	does join	{D}	=>	{*}
(A,B)	does join	{A}	=>	(A,B)
(A,B)	does join	{C}	=>	(A,B,C)

III. *SET with *SET

{A}	does join	{B}	=>	{A,B}
(A)	does join	(A)	=>	(A)
{A,B,C}	does join	{B,C}	=>	{A,B,C}
{A,B,C}	does join	{B,C,D}	=>	{A,B,C,D}
{A,B}	does join	{C,D}	=>	{A,B,C,D}
{A;B;C}	does join	{B;C;D}	=>	{B;C}
(A,B)	does join	(A,B)	=>	(A,B)
{A,B}	does join	(A,B)	=>	(A,B)
(A)	does NOT join	(B)		
(A,B)	does NOT join	(C)		
(A,B)	does NOT join	(C,D)		
{A,B}	does NOT join	(C,B)		
{A;B}	does NOT join	(A,B)		

Table 2: Joins with Atomic sets

I. *GROUP with individuals

[A]	does NOT join	A		
$\langle A \rangle$	does NOT join	A		
[A,B]	does NOT join	A		
$\langle A, B \rangle$	does NOT join	A		
[A;B]	does NOT join	A		

II. *GROUP with single element sets

[A]	does join to	[A]	=>	[A]
$\langle A \rangle$	does join to	[A]	=>	$\langle A \rangle$
[A,B]	does NOT join	[A]		
$\langle A, B \rangle$	does NOT join	[A]		
[A;B]	does NOT join	[A]		

III. *GROUP with *GROUP

[A,B]	does join to	[A,B]	=>	[A,B]
$\langle A, B \rangle$	does join to	$\langle A, B \rangle$	=>	$\langle A, B \rangle$
[A;B]	does join to	[A;B]	=>	[A;B]
[A,B]	does NOT join	[A;B]		
[A,B]	does NOT join	$\langle A, B \rangle$		

```
"{x,y}" and "{x,y}" are compatible
"{p,q,r}" and "{p,q,r}" are compatible
"{{a,b},@1,[x,y]}" and "{{a,b},@1,[x,y]}" are compatible
"{[y,x],{b,a}}" and "{[y,x],{b,a}}" are compatible
"{a;b;c}" and "{a;b;c}" are compatible
"{c;d;e}" and "{c;d;e}" are compatible
"{x;y}" and "{x;y}" are compatible
"{p;q;r}" and "{p;q;r}" are compatible
"[a,b,c]" and "[a,b,c]" are compatible
"[c,d,e]" and "[c,d,e]" are compatible
"[abcd,[x,y]]" and "[abcd,[x,y]]" are compatible
"[[y,x],abcd]" and "[[y,x],abcd]" are compatible
"[a;b;c]" and "[a;b;c]" are compatible
"[b;c;d]" and "[b;c;d]" are compatible
"(*)" and "(*)" are compatible
"(a)" and "(a)" are compatible
"(a,b,c)" and "(a,b,c)" are compatible
"(b,c,d)" and "(b,c,d)" are compatible
"<a>" and "<a>" are compatible
"<a,b,c>" and "<a,b,c>" are compatible
"<b,c,d>" and "<b,c,d>" are compatible

*SET with individuals

"a" and "{*}" are compatible with new ind - {a}
"a" and "(*)" are compatible with new ind - (a)
"a" and "{a}" are compatible with new ind - {a}
"a" and "{a,b,c}" are compatible with new ind - {a,b,c}
"a" and "{c,d,e}" are compatible with new ind - {a,c,d,e}
"a" and "{a;b;c}" are compatible with new ind - {a}
```

```
"a" and "{c;d;e}" are compatible with new ind - {*}
"a" and "(b,c,d)" are compatible with new ind - (b,c,d,a)

*SET with single element sets

"{a}" and "{*}" are compatible with new ind - {a}
"{a}" and "(*)" are compatible with new ind - (a)
"{a}" and "{a}" are compatible with new ind - {a}
"{a}" and "{a,b,c}" are compatible with new ind - {a,b,c}
"{e}" and "{c,d,e}" are compatible with new ind - {c,d,e}
"{e}" and "{a;b;c}" are compatible with new ind - {*}
"{e}" and "{c;d;e}" are compatible with new ind - {e}
"{a}" and "(b,c,d)" are compatible with new ind - (b,c,d,a)

*SET with *SET

"{c}" and "{a}" are compatible with new ind - {a,c}
"(a)" and "(a)" are compatible with new ind - (a)
"{a,b,c}" and "{c,d,e}" are compatible with new ind - {a,b,c,d,e}
"{a,b,c}" and "{x,y}" are compatible with new ind - {a,b,c,x,y}
"{a;b;c}" and "{c;d;e}" are compatible with new ind - {c}
"(a,b,c)" and "(a,b,c)" are compatible with new ind - (a,b,c)
"{a,b,c}" and "(a,b,c)" are compatible with new ind - (a,b,c)
"(b,c,d)" and "(a,b,c)" are incompatible
"{a,b,c}" and "{a;b;c}" are compatible with new ind - {{a;b;c},a,b,c}

*GROUP with individuals

"a" and "[a,b,c]" are incompatible
"a" and "<a>" are incompatible
"a" and "[c,d,e]" are incompatible
"a" and "<a,b,c>" are incompatible
"a" and "[a;b;c]" are incompatible

*GROUP with single element sets

"[a,b,c]" and "[a,b,c]" are compatible with new ind - [a,b,c]
"<a>" and "<a>" are compatible with new ind - <a>
"[a,b,c]" and "[c,d,e]" are incompatible

*GROUP with *GROUP

"[c,d,e]" and "[c,d,e]" are compatible with new ind - [c,d,e]
"<a,b,c>" and "<a,b,c>" are compatible with new ind - <a,b,c>
"[a;b;c]" and "[a;b;c]" are compatible with new ind - [a;b;c]
"[c,d,e]" and "[a;b;c]" are incompatible
```

3.2 Project

Since join and project are duals, we will not waste space by repeating the same tests that we presented for join. The following set of correspondences are sufficient to indicate how project works:

Join	Project
Max. Common Subtype	Min. Common Supertype
Union	Intersection

4 An extended example

As an example of complex processing involving sets, we present a portion of a military intelligence demonstration program[4]. The demonstration involves the generation of enemy intentions based on simple physical intelligence about the position and movement of ground troops. At a particular point in the processing, our reasoning engine, MGR, has generated possible avenues of approach and the estimated time of arrival at each avenue for each of a number of divisions. Since each is an alternative route for the division to take, there is a need for a disjunctive set. Two such sets are as follows (these are extracts from larger graphs – here we use the linear form):

$$[UNIT : D39] \rightarrow (POS) \rightarrow [AA - ETA : \{\langle C, @77\rangle ; \langle [C, B], @96\rangle ; \langle B, @96\rangle ; \langle [B, A], @115\rangle\}$$

$$[UNIT : D4] \rightarrow (POS) \rightarrow [AA - ETA : \{\langle B, @46\rangle ; \langle [B, A], @85\rangle ; \langle C, @131\rangle ; \langle [C, B], @131\rangle\}$$

Note that each triple four alternatives each one of which is an Avenue-ETA pair (a tuple). In some tuples the avenue is a simple name (A, B or C) whereas in some it is a fixed size group of two avenues e.g. [C,B]. The next stage is to break the disjuncts and produce separate graphs from the possible alternatives. The control structure of MGR produces eight graphs of which the following are two:

$$[UNIT : D4] \rightarrow (POS) \rightarrow [AA - ETA : \langle B, @46\rangle]$$

and

$$[UNIT : D39] \rightarrow (POS) \rightarrow [AA - ETA : \langle C, @77\rangle]$$

Since these alternatives only consider the movements of individual divisions, and the plan presumably involves several divisions (including D4 and D39) MGR then joins these alternatives to a schema containing empty lists for both UNIT and AA-ETA i.e.

$$[UNIT : (*)] \rightarrow (POS) \rightarrow [AA - ETA : (*)]$$

Each division is joined separately, which is possible because of the open nature of (∗). Proper order is maintained by the list property of the resultant graph. Thus the join of the two graphs for D4 and D39 separately with the schema produces:

$$[UNIT : (D39, D4)] \rightarrow (POS) \rightarrow [AA - ETA : (\langle C, @77\rangle, \langle B, @46\rangle)]$$

The semantics of the () notation for ordered sets ensures the correspondence between a division name and its AA-ETA.

5 Conclusion and Future Enhancements

We have shown how an analysis of the properties of several kinds of sets leads to a lattice of set types which can act as the basis of an extension to the graph operations join and project to handle simple and compound individuals. Since they also correspond to familiar data structures in programming languages, we can now provide conceptual graph processing systems with a well-founded set of extensions that are useful in a general processing sense, but are also related to conceptual graph theory.
In the future we will further incorporate our notation for sets into our reasoning system MGR, and extend the notation to include the size attribute that we mentioned above. Eventually our conceptual graph system CP, we believe, can be used to support any knowledge-based system work. The set extensions we have described here go a long way towards the fulfillment of this goal.

[4]This work is sponsored by the USAICS, Fort Huachuca under a contract with TRADOC

References

[1] Eshner, D.P. and Hartley, R.T. (1988). Conceptual Programming with Constraints. In *Proceedings Third Annual Workshop on Conceptual Graphs*, Minneapolis, MN, 3.1.2-1-3.1.2-6.

[2] Gardiner, D.A., Tjan, B., and Slagle, J.R. (1989) Extended Conceptual Strcutures Notation. *Proc. Conceptual Graphs Workshop*, Detroit, pp. 3.05-1 - 3.05-11.

[3] Hartley, R. T. (1986). The Foundations of Conceptual Programming. In *Proceedings of First Rocky Mountain Conf. on AI*, Boulder, CO, 3-15.

[4] Hartley, R. T. and Coombs, M. J. (1989) Conceptual programming: Foundations of problem solving. In: J. Sowa, N. Foo, and P. Rao (Eds) *Conceptual Graphs for Knowledge Systems.* New York, NY: Addison-Wesley.

[5] Pfeiffer, H. D. (1986). Graph definition system. *Proceedings of the Second New Mexico Computer Science Conference*, Las Cruces, 97-103.

[6] Pfeiffer, H.D. and Hartley, R.T. (1989) Semantic additions to Conceptual Programming. *Proc. Conceptual Graphs Workshop*, Detroit, pp. 6.07-1 - 6.07-8.

[7] Sowa, J.F. (1984). *Conceptual Structures.* Reading, MA: Addison Wesley.