Presentation

Deutsch: Presentation is intended to encompass notations and languages for expressing models. This session will focus on the linguistic and notational choices made in particular approaches. Emphasis will be placed on common ideas. For example, there have been some assertions from proponents of the predicate calculus that it is a notation that is capable of expressing essentially all the interesting and important concepts that are encountered in other notations. Emphasis will also be placed on why the design choices were made, why things are being represented a certain way, and what the effects of those choices were.

The purpose of this workshop is to try to bring us all closer together. Trying to bring us to a common terminology is something I don't have much hope for at this point. However, getting us to recognize in other areas problems and solutions we have encountered in our own work is something that I think is very possible, and I think it has happened already to a certain extent. I see that as a goal of this session.

The main body of the session will consist of short presentations that emphasize linguistic or notational choices. I will begin by offering the following taxonomic framework as a structure for comparing the various approaches.

- Levels of viewpoint - I think there are useful distinctions to be made about whether the language claims to be discussing objects in the real world, or claims to be discussing representation of those objects in the machine, or the operational organization of the system. In my opinion, AI has made the biggest attempt to distinguish between these levels. In PL systems there has been more of an attempt to disassociate them, that is, the choices made by the language designer have tended to be in a different realm from the choices available to the program writer. In the DB world I see a trend from keeping them separate towards merging them together, that is, treating what has been called a data dictionary in a way more similar to the rest of the data represented in the data base. So, when someone starts talking about their language or notation, they might say something about which of these levels they are addressing.
- Target population - Different languages are addressed to different kinds of users: system designers and builders, application designers and builders, system instance administrators, end users.
- Purpose of notation - There are four kinds of emphasis, not necessarily disjoint, that one finds in different kinds of notations: specification, implementation, validation and communication. One might want to say something about where the emphasis lies in their notation.
- Underlying formal structures - I happen to be of a school that believes that mathematical structures can be sound formally. So, I encourage people to talk about the mathematical structures underlying their notations. Here are some examples: predicate calculus, algebras, sets and mappings, rewrite rule systems.
- Procedural/descriptive emphasis - This used to be a tremendous source of argumentation in the AI community. There is the question of whether a given language is focussing on describing data, or procedures or relationships among things. Of course, every language will have to deal with all three to some extent.
- Primitive constructs, combining forms - Here my imagination failed me somewhat. But, some examples of primitive constructs are: various kinds of variables and constants, primitive operations, primitive functionals. Some examples of combining forms are: functional composition, type constructors, mapping/function/relation constructors. There may not be a sharp distinction between the primitive constructs and the combining forms.
- Abstraction relationships - I have taken the liberty of extracting some various kinds of abstraction mentioned in Michael Brodie's paper. These are: generalization/specialization, classification/instantiation, aggregation/decomposition and association. There was a fifth one whose meaning I didn't understand, and therefore I left it out.

These are some dimensions along which people might find it helpful to classify their languages and constructs. The remainder of the session will consist of short presentations by Charles Rich, Ray Reiter, John Sowa, Jaime Carbonell, Martin Feather and Pat Hayes.

THE PLAN CALCULUS

Rich: The notation that I am going to talk about is a diagrammatic formalism that we have developed in the Programmer's Apprentice project at MIT. The notation has mostly been used to represent programs and abstractions of programs, so it is a program design language. The notation is called "Plan calculus". One purpose of the notation is to capture a collection of programming cliches that can then be used as input to a program transformation process to solve a given problem.

I am going to explain the primitive types, the ways of composing them and, finally, the notation that we use for abstraction. There are basically three primitive types: (1) IOSpecs, whose instances are called operations, (2) Test Specs, whose instances are called tests, and (3) Primitive object types such as integers, sequences, atoms, etc.

The basic compositional method that we use is the gathering together of a set of named parts and constraints on those parts. In particular, if the parts are all primitive objects or compositions of primitive objects, the resulting composite type is called a data plan. Instances of data plans are data structures. A simple example of a data plan is a "pair" where the two parts are called "left" and "right". An example of a data plan with constraints would be an ordered pair, where the left and right parts are both numbers, and the left part is less than the right part. The composite type consisting of both data structures and input/output or test specifications is called a temporal plan. Instances of temporal plans are computations, things that include time and data structures.

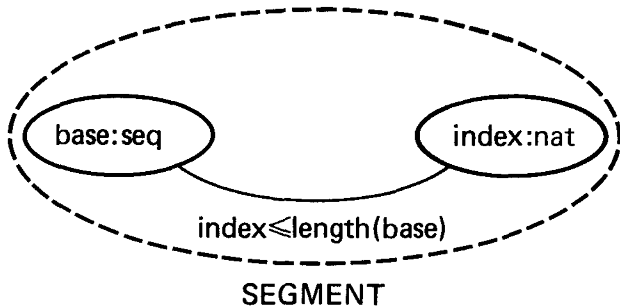
The final notion I will talk about is called an overlay. Overlays may be used to represent different points of view, such as the relationship between an abstract description and an implementation of it, or program transformation relationships between two types of temporal plans. An overlay is a triple, consisting of two plans and a correspondence between them.

Levesque: Is a data structure an example of a data plan? (Rich) No. It is an instance. It is a type/token relationship. (Levesque) What is another instance of a data plan? (Rich) Each data plan is a type, and each instance of the data plan is an instance of that type. I may have many different types. "Triple" is also a data plan. A particular triple like <1, 2, 3> is a data structure.

A Diagrammatic Formalism

Rich: The presentation of plans is an important aspect of the Plan Calculus. We have found it very useful to have a diagrammatic formalism for representing (abstract) plans. The system stores the plans in an encoded form, but we found a more topological presentation easier to work with.

An example of a data plan for a type called SEGMENT is shown in the following figure (Rich-1).



SEGMENT

Rich-1

SEGMENT has two parts, called base and index. The type of base is sequence and the type of index is natural number, and there is a constraint between the parts that the index is less than or equal to the length of the base. An instance of this data plan would be a particular sequence. This type might be used to implement stacks or queues.

Thatcher: Is there any way in your notation that "nat" could be replaced by "T"?

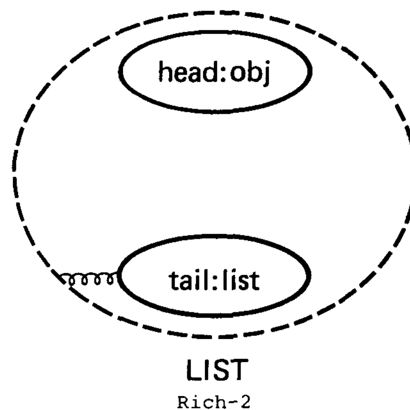
Rich: Yes. There is a type hierarchy. In fact, in the data plan in figure Rich-1, "object" is used as the union (type) of all the primitive types. So, it is not quite parameterization, but there are various specialization relationships that will allow you to talk about a list whose head is a list of integers, in which case you would say that the head is specifically an integer.

Figure Rich-2 shows a list which is a data plan having two parts, head and tail. The head is an object, the tail is a list and there are no other constraints.

Hitchcock: What would this actually look like in machine-readable form?

Rich: That's totally implementation dependent and could be done three or four different ways.

Hardgrave: Is there any distinction between the fact that you've got "list" inside the box in lower case letters and the "LIST" outside the box in upper case?



LIST

Rich-2

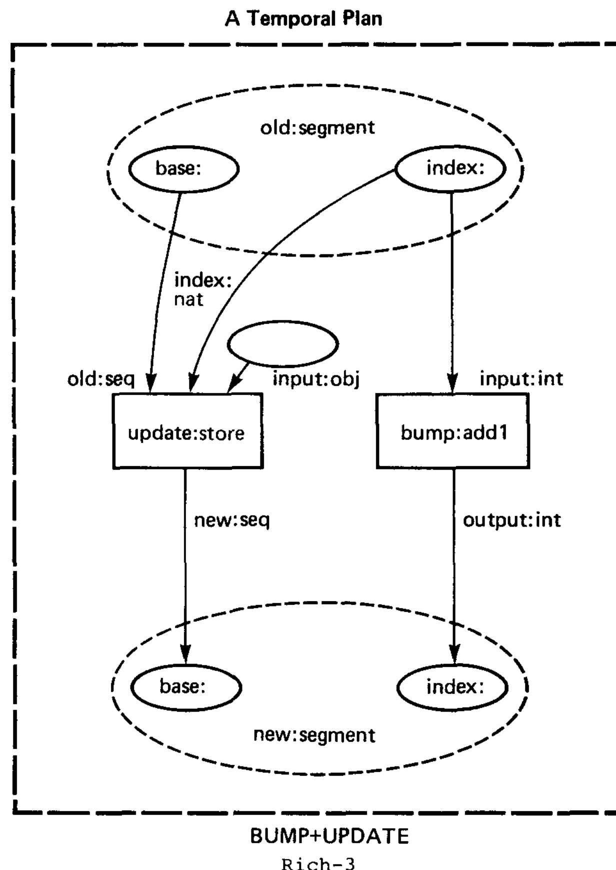
Rich: No. In fact they are the same. The case is not important. What is important is that this is a recursively defined type. In fact, there is a bit of notation we use for recursively defined types. We denote the recursive part by the little squiggle to the left of tail, which is the symbol for induction in electronics.

Thatcher: Can you tell us what goes inside the dotted lines. It looks like what I would call a signature but with operations and what arguments they take and what values they produce.

Rich: Intuitively, these part names are functions of the objects. They are the primitive parts.

Figure Rich-3 is an example of a temporal plan for BUMP+UPDATE.

A Temporal Plan



BUMP+UPDATE

Rich-3

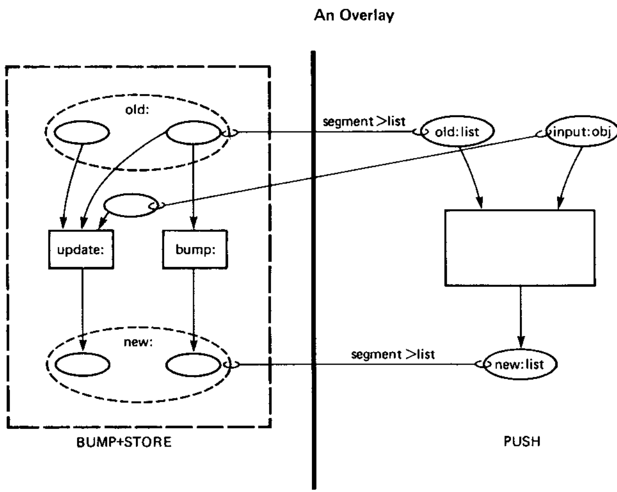
Let me go through it because there is a fair bit of detail. This plan captures a programming cliche that

takes an array and a pointer, stores a new element at the place pointed to and adds one to the pointer. The square boxes are operation boxes, and the arrows coming in are inputs, and the arrows coming out are outputs. There are two uses in this diagram of the SEGMENT example we saw before. At the top level this diagram shows four parts, named "old", "update", "bump", and "new." Each part has a type: old is a segment, update is a store operation, bump is an instance of add1, and new is a segment. The important constraints here are data flow and control. There aren't any examples here of control flow, because that really only becomes important when you have tests. The arrows are data flow arrows. The intuitive idea is that the base sequence of the old segment is input to a store operation, and the output of the store operation becomes the base sequence of the new segment. The store operation has three inputs: the old sequence, the index, and the object to be stored.

Overlays

Rich: Figure Rich-4 is an example of an overlay. An overlay is an expression of a relationship between two different points of view. Each point of view is represented as a plan. This figure is an example of an implementation point of view. The plan on the right-hand-side is a plan for a push operation. It has three data parts: the old list, the input (which is the object to be pushed) and the new list which is output. On the left we have the plan from Rich-3.

An Overlay



Rich-4

The overlay specifies the correspondences between the two diagrams [i.e., the links crossing the center line]. This overlay says that an instance of the plan on the right can be viewed as an instance of the plan on the left via the indicated correspondences. The input to the update part of BUMP+STORE corresponds to (can be viewed as implementing) the object being pushed onto the list in PUSH. This is a straight equality correspondence. Similarly, the old segment can be viewed as implementing the old list. The blank box on the right-hand-side of the diagram represents another overlay, not detailed here. It would be a formal expression of the implementation of a list using a sequence and a pointer.

Every overlay has a name. BUMP+STORE>PUSH is the name of this overlay. The correspondence between segments and lists is specified by SEGMENT>LIST which is used to build the BUMP+STORE>PUSH overlay. Naming allows an overlay to have multiple occurances in a plan. SEGMENT>LIST is used twice: once to show that the old part of BUMP+STORE can be viewed as the old list according to this overlay, and again to show the the new part of BUMP+STORE, which is also a segment, can be viewed as a new list also according to the same overlay. Basically, overlays are used to represent all relationships between plans, other than specialization.

Balzer: Isn't there an additional constraint in the fact that segment does have to be the same down below as it would be used as input to the plan?

Rich: Well, no, because on the right-hand-side the old list and the new list in this particular formulation do not have to be the same. This is a very abstract PUSH. You are getting a new list out. You can have a more refined version of push to which you can add the constraint that the old list is identical to the new list. In that case you would add the constraint that this sequence is equal to that sequence and you would have a specialized version of the whole overlay.

A LOGICAL VIEW

Reiter: I am going to try to convince you of the virtues of predicate logic as a language for the representation of world knowledge. The pun in the title of this talk is intended. This figure, Reiter-1, illustrates how first order logic (FOL) can be used to represent a database.

Variables: x,y,z,x1,y1,z1,...
Constants: a,b,John-Doe,ship33,...
Predicates: SUPPLIES(•,•),EMPLOYEE(•),...
Logical Constants: $\Rightarrow, \wedge, \vee, \neg$
Quantifiers: $\forall x, \exists x, \forall y, \exists y, \ldots$
Well Formed Formulae:
SUPPLIES(Acme,ship33)
EMPLOYEE(John-Doe)
$\forall x$[EMPLOYEE(x)$\Rightarrow$HUMAN(x)]
$\forall x,y,z$[SUPPLIES(x,y)$\wedge$SUBPART(z,y) $\Rightarrow$ SUPPLIES(x,z)]
$\forall x$[EMPLOYEE(x)$\wedge$MALE(x) $\Rightarrow$ $\neg$PREGNANCY-BENEFITS(x)]

Reiter-1

We start with variables, constants, predicates, logical constants, and quantifiers. Then, there are some examples of well-formed formulae. Finally, a data base is defined as any finite set of formulae in first order logic. Clearly, very few of us in this room are prepared to implement a system for such a totally unstructured notion of data. So, what we would typically want to do is to introduce a variety of data models which will put suitable restrictions on this completely unrestricted notion of "data base".

D. Smith: What is the intended purpose of this kind of notation?

Reiter: The purpose is to model the real world somehow. The audience is all of you.

D. Smith: Would you show a specification like this to an end user who wants to write a query against that data base?

Reiter: No. I would not want to show anything like this. This would have to be appropriately sugared up with fancy syntax. But I do not see that as a deficiency. The predicate logic form is the underlying representation language. It is quite possible to reformulate the language in a syntactically nice, sugared way for end users. I am not really addressing that issue.

The Adequacy of FOL

Hitchcock: Is it adequate, in fact, to express what you want to express? I am thinking of the predicate "ancestor", which is a transitive closure, which is not a first-order operation.

Deutsch: You cannot represent generic transitive closure, but you can represent any particular transitive closure.

Reiter: Right. I can represent transitivity of any particular relation. For example, I can say that if x is the ancestor of y and y is the ancestor of z, then x is the ancestor of z.

Hitchcock: The problem with ancestor is it can entail many generations.

Reiter: You can recursively apply that again.

Sowa: You can define the ancestor of human ancestor, but you can't define a general ancestor operator that will take any transitive relation and create a new ancestor of that. So you can't create ancestor in first order predicate calculus. However, for any particular relation you can.

Hardgrave: Dont you also need set theory?

Reiter: No, I am using FOL to represent the content of the data base. I am not trying to give the definition of "data base" itself within first-order logic.

Hardgrave: But, you will need set theory to manipulate sets of objects.

Data Models as Restrictions on FOL

Reiter: This completely unstructured notion of database is altogether too rich for any reasonable system implementation. What is clearly required is some appropriate notion of restriction that we can apply to a class of data bases, so as to come up with computationally reasonable implementations of database systems. Data models then simply correspond to imposing appropriate requirements on the class of data bases which will be investigated. Let me give a few examples of how we can impose requirements so as to restrict the class of databases we want to consider.

Types

Reiter: The first thing I want to mention is the notion of type, and how we can introduce it into this framework that I have defined. A type is simply a distinguished unary predicate. You just have to decide what unary predicates you want to distinguish and then call them types. Examples are EMPLOYEE, SUPPLIER, and so forth. Having fixed on a set of types, then within this formalism you can represent most of the things you really would like to. It is simple to represent subtypes using implications, as in

$\forall x[\mathrm{EMPLOYEE}(x) \Rightarrow \mathrm{HUMAN}(x)]$.

Disjointness can be represented, as in

$\forall x \neg [\mathrm{MALE}(x) \wedge \mathrm{FEMALE}(x)]$

And new types can be defined as arbitrary boolean combinations of old types, as in

$\forall x[\mathrm{BACHELOR}(x) \equiv \mathrm{MALE}(x) \wedge \mathrm{SINGLE}(x)]$

So you can build up a type hierarchy in a very straightforward way.

Balzer: I don't understand how this places a requirement on a database, how it places a restriction on what is possible to do.

Reiter: It really hasn't yet.

Rich: More than that, how does it make things more efficient. That's the whole game, right?

Reiter: Efficiency would be purchased in part by imposing certain restrictions on how the types are to be used. I will come to this point a bit later.

Thatcher: Are you going back seven years and saying that types are sets?

Reiter: No, I am saying that types are unary predicates. I am not identifying a type with the extension of a unary predicate.

Buneman: Can I place an object in the database and get some attributes that won't have a type?

Reiter: Yes. If you want to insist that all objects have type, that places further restrictions on the classical outlook. You are further defining the data model.

Rich: How do you say that for any new type predicate you introduce, that it is a subtype of some existing type?

Reiter: I can say that as a statement about the data model. I can simply require as part of the specification of the data model that every individual must be an instance of one of the types.

Other Examples of Data Model Restrictions

Reiter: Let me give you an example of how a data model can support the concept of a relation. Let us assume we have some types and that we now want to talk about relations. The way to do that is to impose the requirement on the database that for each non type predicate R there is a formula in the database of the form

$\forall x1,\ldots,xn[R(x1,\ldots,xn) \Rightarrow D1(x1) \wedge \ldots \wedge Dn(xn)]$

where R is a relation and D1,...,Dn are its domains. An instance of a relation is some tuple of constants such that you can prove the relation within first-order logic using the database as premises. The extension of a relation is the set of its instances.

Two other restrictions leading to different data models are the following. First, the closed-world assumption says that if you can not prove a relation from a data base then you are permitted to infer the negation of that relation. Second - and this comes back to how do you ensure a certain form of efficiency with respect to type - you might want the set of types to be extensionally complete. This means that there is a decision procedure for type membership. But the existence of this decision procedure can be formally defined using strictly logical concepts.

I am not proposing yet another data model. Instead, I am proposing a language, namely FOL, with which to represent facts about the "real world": a representation language. The advantages of FOL are:

1. It is a very expressive language.

2. It has a well-defined denotational semantics and therefore we can set up a correspondence between the constants and predicates of the language and "real-world" individuals and relations.

3. It has a proof theory, which provides for inferential retrieval of answers and provides a reference standard for the deductive capabilities of various data models (as the relational model provides a reference standard for the relational view).

4. It provides a reference standard for comparing relative powers of different nonlogical data models.

Shaw: I would like to ask what kinds of restrictions you incur by choosing to remain within first-order logic? For example, point 4 seems to imply that you can take any data model and express it here. Could you characterize some of the limitations?

Reiter: I would not like to make so strong a claim that all non-logical data models map into this framework, but I suspect that a lot of them do. When they do, then we are in a good position to compare. I would guess that any data model that uses relations as objects will map in, in a very straight forward way.

Goldstein: The thing that disturbs me, and I know you are sensitive to it, is the set of issues that has arisen about the sufficiency of first-order logic as a basis for problem-solving. In particular, the standard issue raised is the control of inference. That arises in the database world as well. Certain uses of the data model make certain kinds of retrievals more efficient. I wonder how you would address that set of questions given the first-order logic formalism.

Reiter: Are you asking how is the inferencing controlled? Or are you asking how do I sufficiently delimit the class of data bases by a data model such that the inferencing within that limited class will be efficient?

Goldstein: I'll settle for the second.

Reiter: That's a serious research problem.

Goldstein: Let me sharpen my criticism. It doesn't seem to help much to observe that you can express what you have expressed in first-order logic. The game here seems to be one of efficiency and not one of logical completeness of the database.

Reiter: I am not precluding data models that are incomplete. There will be lots of questions within those data models which simply can't be answered.

Thatcher: I really am all for your 1, 2, and 3 but then you lost me because you started using the term "data model" and you didn't tell me what it was. What is it?

Reiter: I suggested that a data model imposed requirements on a database. I am not specifying any particular data model. I am providing you with a conceptualization as to how to define a data model within this representation formalism.

Shaw: Is a data model then a set of restrictions, not necessarily first-order, imposed on this set of unary predicates.

Reiter: Any set of restrictions that you can articulate in English cum mathematics is possible.

Thatcher: Are you saying that a database is a set of formulas, and a data model is somehow restricting the possible formulas that may occur in the database representation.

Reiter: That is one way of defining a data model, but I gave a couple of examples that did not have that character. For example, imposing the closed-world assumption.

CONCEPTUAL GRAPHS

Sowa: I am going to talk about conceptual graphs. The notation has two basic principles: a concept, which is a box containing a type label, and a conceptual relation, which is a circle with a label in it. A conceptual graph is a graph consisting of boxes and circles attached together by arrows. The figure, Sowa-1 shows a simple conceptual graph for the sentence, "A person feeds Morris."

Sowa-1

Most of the relationships are dyadic, although other numbers of arcs occur also. You can have a set of builtin, primitive type labels and relation labels, or you can define new ones through a method of lambda abstraction.

Concept boxes represent things. Inside a concept box you may find a generic type label, such as PERSON, representing any person. Alternatively, you may find a type lable, individual identifier combination, such as CAT:Morris, representing a particular cat. Quantification over a type is specified by replacing the individual identifier with a quantifier symbol, such as CAT:∀ (for all cats) or CAT:∃ (there exists a cat). Queries can be represented by using a "?" in the individual identifier position, such as CAT:? (give me a cat).

Some examples of conceptual relations (circles) are the agent (AGNT) relationship which specifies the actor in an action, and the patient (PTNT) relationship (which is used in preference to the more common object relationship) which specifies what is acted upon. Besides the linguistic case relations examplified above, one can also have mathematical and other semantic relationships, such as the RESULT of a mathematical operation.

The type labels are the basic categories, which are ordered by levels of generality. So, you have a type hierarchy: MANAGER is a type of EMPLOYEE, EMPLOYEE is a type of PERSON, etc. There is a relation that tells which individuals belong to which type. We use the notation t::i to say the individual i is of type t. There is a special "undefined" marker "@", representing "any". The marker @ conforms to all types. For any type, the referent to a concept (i.e., an individual of that type) must conform to that type.

There is a set of user-defined graphs that are well-formed by definition. This is called the canon. The canon has background knowledge and it has a set of type labels with a partial ordering defined over them. You can extend the type ordering to a type lattice if you choose, but all that is assumed is a partial ordering. There is an equality relation between the individual markers and the types. This canon is a basis set of graphs, a basis from which you can derive all the others. There are four formation rules, which give you a kind of graph grammar. The formation rules are: copy, detach (any conceptual relation), restrict (any type label to a subtype), and join (two graphs about a common concept node). The transitive closure of the canon under these four operations defines the set of well-formed conceptual graphs. Basically, all of predicate calculus can be represented in the form of conceptual graphs (using some conventions not described here). [See the IBM Journal of R & D, Vol. 20, No. 4, July 1976, pp. 336-357 for further details on conceptual graphs.]

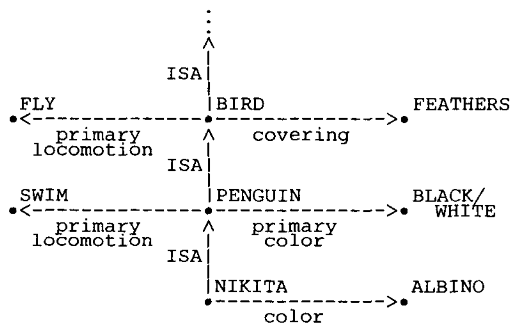
Conceptual graphs represent information. Computations over this collection of information are defined in a transition notation. The transistion notation is based on Petri Net notation. In it, a transition graph is superimposed on a conceptual graph. The transistions in the transition graph take their inputs from and produce as outputs values associated with the concepts of the conceptual graph. More than one transition graph can be superimposed on a single conceptual graph; for example, the addition relation, a+b=c, can be used to compute c from a and b or to compute a from b and c (a subtraction).

Computations are triggered by the presence of the control mark "?". A second control mark, "!", is used to assert values to be used in computations. The computation is controlled by the propagation of these marks thru the transition graph.

Deutsch: Giving multiple computational interpretations to a single assertion is similar to the work of Sussman and his colleagues at MIT on "constraints". [See also Deutsch's position paper in this proceedings.]

INHERITING PROPERTIES

Carbonell: I am going to talk about the problem of inheritance, representing inheritance, the semantics of it. This work is primarily on semantic networks in AI. Figure Carbonell-1 is an example of a semantic network.



Carbonell-1

It shows that NIKITA is a penguin, penguins are birds, a bird's primary mode of locomotion is to fly, etc.. One of the main points of the networks is the inheritance of the ISA links. Since birds have feathers, penguins have feathers, and since Nikita is a bird, Nikita has feathers. You can also state exceptions. A penguin's primary mode of locomotion is swimming; this overrides the otherwise inherited default, flying. So this is not strict ISA inheritance. I want to complain about the use of ISA links here. We need a much better notation and semantics of what ISA links are.

There are three types of inheritance, DOWNWARD, UPWARD, LATERAL.

Downward Inheritance

Carbonell: Downward inheritance is what we have along ISA links. ISA has also been called superclass, set-inclusion, etc. When you say "A is a B", it could mean set inclusion, membership in a set, or non strict inheritance. I am proposing downward inheritance, where everything that is true about B will be true about A. You give a function that tells you which attributes associated with B are to be associated with A. Not all properties may be inherited. Some may be filtered out. Also, some attribute names may change their names. The function specifies the correspondence of these as well.

Upward Inheritance

The second form of inheritance is what I call upward inheritance. Suppose we have a PARTOF relationship; for example, "B is PARTOF A." Further suppose B is a typical part of A. Then A can inherit certain properties from B. That is, some of the attributes of A are really attributes of the parts B_i of A.

When A is a collection of objects $\{B_i\}$ all of the same type, then a second kind of upward inheritance can be defined. The unit A can inherit attributes which are combinations of the attributes of the B_i's of which A is composed. As an example of this second kind of upward inheritance, suppose we want to know the population of South America, and this information is not stored explicitly in the database. But, when we look at all the parts of South America (for example, the geographical divisions such as Countries), each one has a population. How do we know that calculating the population of the whole continent from the individual countries is a valid deduction? If we know that the covering by the countries of the continent is complete and that the division into countries is a partition, then we know the deduction is ok.

Balzer: So in general, you have to know what the rules of combination are.

Carbonell: That's right. And I have just given you the default rules. The rules of combination should be specified in the data base as an explicit kind of self-knowledge.

Lateral Inheritance

Carbonell: The last kind of inheritance, which I call lateral inheritance, is where I do things such as exception handling. I wanted to have a strict inclusion hierarchy, and in addition be able to have things like exception handling. I want to be able to say "X is like Y except ...". I also want to represent things like "toy car" or "fake gun". You could represent a toy car by starting from scratch. But somehow your knowledge of what a car is should allow you to state what a toy car is much more economically. It is useful for "toy car" to inherit most of the information from "car". This is lateral inheritance. It does not imply inclusion of types. You can specify what properties should go through and specify as inputs which other things can override specific properties.

What I really have is a notion of semantic inheritance. This sort of thing makes inheritance explicit. It has been useful in AI applications, at least in mine. I am really not sure if it has applications in DB and PLs.

Shaw: Inheritance has a direct analog in PLs. Suppose I have a definition of "deque" that has operations such as insert-front, insert-back, delete-front, delete-back. Now I would like to derive from that a representation for a queue in the obvious way - by deleting two functions and renaming the other two. It seems to me that is directly analogous to your notion of downward inheritance

Balzer: I thought upward inheritance.

Carbonell: Perhaps the best way to do that is by lateral inheritance.

Shaw: My point is that there are often situations where you want to define one type as a slight variation of other types.

Balzer: This seems to be an issue related to what Chuck Rich was talking about. Namely, the problem Mary has of finding the correspondence between two specifications or representations seems to be the same as: if you do not see locally the answer to some question, then this tells you where to go - you may go up, or down or sideways.

Rich: Your lateral inheritance is like my "overlays", i.e., a function from one domain to another domain that gives correspondences between attributes - I call them parts. I have gone farther and have used this for representing implementation relationships, points of view, etc., not just for talking about penguins and birds.

Deutsch: I find it very exciting that Jaime, Mary, and Chuck, who presumably have not talked together in any significant way before, find these similarities.

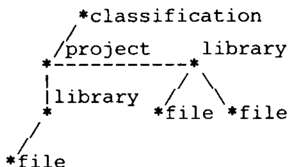
SPECIFYING THE BEHAVIOR OF TYPES

Feather: The approach I want to talk about is an approach to specifying the behavior of types. The overall organization of a specification is composed of four parts which

1. define types and relations to model objects and relations of the task domain.
2. express some "static" constraints that specify what must be true in every state of our world.
3. model changes to the world. We want implicit and explicit invocation, non-determinism, and the full power of historical reference.
4. specify the "dynamic" constraints, those which refer to more than one state.

By the way, we have a motto: if it moves, it's observable. Any change in the world that is observable can be referred to.

Some examples of the kinds of types we want to be able to define are shown in the following figure.



The relationships among types are not necessarily binary. To define a type, such as "project" with attributes "library" and "classification", we write

```
    type project(library,classification).
```

Once defined, there are expressions for refering to the project and its attributes. If p is a project, then the expression "p:classification" denotes the classification attributed to p, and "p:library" denotes a library attributed to p. If l is a library, "then l::library" denotes a project having l as an attribute.

As one defines these types and relations, the reference notation can be used to conveniently express a set of restrictions. For example, you might want to specify "every library is attributed to one and only one project"

```
    type project(library::unique,...),
```

or "every project has precisely one classification attributed to it".

```
    type project(...,:unique classification).
```

The number of occurances allowed for attributes can be specified using the predefined restrictions: any, unique, multiple (more than 0), optional (0 or 1), n (an integer), [n,m] (a range).

Not all constraints can be expressed as above. For example, "all lines in units must be of length card-length" needs the construct always requires.

```
    always require
    ∀line (line insequence *any*:unit-contents
            ⇒ length(line:chars) = card-length)
```

We also can express that changes in the world don't violate constraints. Here is an example: a unit may not be deleted if it is included anywhere. We use the always prohibit feature.

```
    always prohibit
        ∃ed unit, ∃line
            (destroyed(unit)
                ∧old(line:includes-unit=unit))
```

where ∃ed refers to a previous state in which "unit" existed as does old.

McLeod: Does this all apply to relations other than binary?

Feather: We have a notion for general relations, but it isn't as concise as that. It would display types, relations and other constraints all in one thing. Very often that is what you would like to do.

McLeod: Could you please comment on the intended use?

Feather: The intended use is to specify behavior separate from giving the details of implementation. We'd like to see these specifications, along with perhaps some comments on how efficient it should be implemented, given to a transformation system, and then with user assistance, we would work towards a reasonably efficient implementation of that specification.

Schmidt: There are a couple of tools around for the specification of programming languages. For example, denotational semantics. It seems that some of the things related to defining the semantics of language are pretty close to things you define. For example, types and semantic domains, static functions and dynamic constraints. Does anybody have any experience in applying this kind of formal semantics technique from the programming language area to the purpose you have in mind?

Feather: Not us. Really, our inspiration comes from natural language.

FOL AS A NOTATION

Hayes: This is really a footnote to Reiter's presentation. I want to talk about first-oder logic (FOL) as a notation. I think it is even better than Ray thought it was. I think a lot of notations that have been invented, certainly in AI, are variations on first-order logic. They just add sugar which is important, but, still, it is just sugar. An example of such a sugared notation is illustrated in the KRL fragment shown in figure Hayes-1a (N. B., any syntactic mistakes are mine):

```
    UNIT traveller
        self ISA person
        homeport <string>
        age [CASE THE age FROM person(thisone):
            <10: INFANT
            ≥10,<13: CHILD
            ≥13: ADULT]
        destination <port>
```

Hayes-1a

KRL is a notation developed at Xerox by Danny Bobrow in which you talk about objects, called units, slots, an ISA hierarchy, etc. It is a very complicated language. It does not look at all like logic, but it is. As can be

seen from the equivalent set of assertions in FOL shown in Hayes-1b.

$$
\begin{aligned}
&\forall x\, \text{Traveller}(x) \rightarrow \text{Person}(x)\\
&\quad \wedge \text{String}(\text{homeport}(x))\\
&\quad \wedge \text{Port}(\text{destination}(x))\\
&\quad \wedge [\text{calage}(x)<10 \rightarrow \text{travage}(x)=\text{INFANT}\\
&\qquad \wedge 10 \leq \text{calage}(x)<13 \rightarrow \text{travage}(x)=\text{CHILD}\\
&\qquad \wedge 13 \leq \text{calage}(x) \rightarrow \text{travage}(x)=\text{ADULT}]
\end{aligned}
$$

Hayes-1b

I had to do some minor alterations. There are two notions of "age" here. There is the age of somebody being considered as a mere person, and the age of somebody being considered as a traveller. A person's age is a number and a traveller's age is either infant, child, or adult. I had to use calage for calendar age travage for traveller age. With that minor variation, it transcribes right over. Moreover, not only does the notation transcribe, but the proof theory transcribes. That is to say, operations that are performed on this notation in that system turn out to be first order, almost entirely. With the exception of default reasoning, I have yet to see a representation language put forward in AI that is not a syntactic variant of FOL.

Goldstein: You made a remark that the proof procedure goes over directly. Did you mean that everything provable in one is provable in the other or that the complexity of doing the proof is the same in both systems. (The time to do the proof is the same.)

Hayes: That is a complicated issue. I do not want to claim that if you shove those axioms into a FOL theorem-prover that it would do what KRL did. If you look at what KRL did, every single procedural step (they call it matching, I believe) was an instance of first-order inference, and could have been reproduced by doing ordinary first-order unification and resolution techniques. Now the search space of possible inferences is much larger - there's a lot of irrelevant stuff that you could infer from this that KRL didn't. That is a very important computational point. For the moment, I am talking about expressibility.

Rich: I am bothered by this traveller example. In Hayes-1b, there is something lost in the translation from the KRL. Those two predicate symbols, travage and calage, are totally unrelated in Hayes-1b, whereas there is a polymorphism of some sort which is captured in the KRL version.

Hayes: I could easily introduce a relation that relates the two ages:

$$\forall x,y\ [\text{Age}(x,y) \equiv \text{calage}(x)=y \vee \text{travage}(x)=y]$$

Is Second Order Necessary

Hayes: With respect to the first-order versus non-first-order controversy, the expression Hayes-2a looks like second order, but it is just syntactic sugar for the expression Hayes-2b.

$$\forall R\,[\forall x \exists y\, R(x,y) \rightarrow \exists f\, \forall x\, R(x,f(x))]$$

Hayes-2a

$$
\begin{aligned}
&\forall R\,[\forall x\, \exists y\, \text{Apply2}(R,x,y)\\
&\quad \rightarrow \exists f\, \forall x,z\, \text{Apply1}(f,x,z)\\
&\qquad \rightarrow \text{Apply2}(R,x,z)]
\end{aligned}
$$

Hayes-2b

It is really higher-order if and only if you are quantifying over all relations and functions that could possibly be defined by lambda abstraction from the full language. That is a helluva big set. Most people are very rarely quantifying over that set. In particular, in Jaime's talk, when he wants to say "this thing has all the properties of that thing except...", there is a finite set of properties there that need to be transfered. You can translate this over to FOL, no problem at all. You can regard that merely as syntactic sugar for the hideous first-order formula. That is a uniform translation.

Deutsch: There is something that I am a little unclear about. Presumably, you want some relation symbols to appear explicitly in some formulae, i.e., to be the ones that are visible. If you want to express that fact, then at some point you will need some axioms that say, for example,

Apply2(R,x,z) iff R(x,z).

That works fine for relations. But if you want it to work for functions, won't you have to introduce equality?

Hayes: Yes, you might well have to use equality. Your question touches on a deeper issue, which I apologize for not having brought up myself. To do this properly, you really need to introduce a little something outside the regular first-order machinery. But it isn't a different logic; it is just another piece of machinery. It is the stuff Richard Weyhrauch has investigated very thoroughly, in which you have to regard some of your stuff as being about the theory you already have, at a metalevel.

Rich: Peter, I thought you could formalize FOL either relationally or functionally; that is, relations plus equality gives you functions.

Hayes: Yes, there is a tradeoff there. Equality is such a useful thing you have to have it anyway.

Thatcher: What you are saying can't be true. Take the Peano axioms for arithmetic, which only involve one second-order sentence, which is the induction axiom.

Deutsch: That would be the same as getting all the functions you could make by lambda abstraction.

Thatcher: That is exactly my point.

Hayes: That is why it is so powerful. It is not reducible to any of its first-order instances. What I am saying is this statement in Hayes-2a looks like second-order, but it need not be. You can interpret that several ways.

Thatcher: But to interpret it one way, I have to write down axioms to say what APPLY1 and APPLY2 mean.

Hayes: Well, it might be infinitely tedious, but it can be done. Then you can do all your first-order reasoning within this odd way of expressing yourself shown in Hayes - 2b. You can transcribe the whole of your theory into this notation.

Thatcher: Not mathematically, I can't. No way.

Rich: The axiom of induction is not part of the axioms of arithmetic. To get the numbers out of the axioms of arithmetic, which are first-order, you need to apply the principle of induction, and that principle itself is not expressible in first-order. Isn't that the issue?

Hayes: I can write down Peano's induction axiom and I can interpret it this way, first-order as in Hayes - 2b, but then it is not Peano's induction axiom anymore. But I

can still do this translation. Look! I am just trying to make a trivial point. Because I want to say something is true of every relation in some set, and I want to write it this way (Hayes-2a) that doesn't necessarily mean I am not within a first-order theory. That is the only point I am making. In many of the applications in AI where the things you want to say look like that in Hayes-2a, one should not conclude they are outside of first-order. Although it looks like quantification over all relations, it almost never is, in fact. It is almost always a finite and explicitly defined set.

Hitchcock: What if it is expressed that one string is a permutation of another? It is not possibly first-order.

Hayes: That is clearly false. This brings me to my next point. I can translate that into FOL. This is well-known. Kowalski has written a book on this and there are many papers. Thank you. You got me nicely to my next point.

I want to object to the procedural/descriptive distinction. Really, it is time we stopped making that distinction. Here is a program.

```
REVERSE(x,y) = valof(test x=nil then
                        resultis y
                        or (y:=cons(cdr(x),y)
                            x:=car(x) )
                        repeat )
```

Hayes-3a

It is approximately in BCPL. It is the usual iterative list reversal. In Hayes-3b it is written in LISP.

```
(DEF REVERSE (X,Y) (COND((NULL X) Y)
                        (T
                        (REVERSE (CDR Y)
                          (CONS (CAR X) Y)) )) )
```

Hayes-3b

There are those who would claim that iteration and recursion are fundamentally different. I do not believe that. Certainly, it is well known you can get the iterative algorithm from a tail recursion by making little remarks about sharing storage.

Rich: There are actually interpreters which will run the second algorithm the same as the first.

Hayes: Now, in Hayes-3c, the same algorithm is written as a pair of equations in FOL.

```
∀Y REVERSE(NIL,Y) = Y
∀Y,A,B REVERSE(A<>B,Y) = REVERSE(B,A<>Y)
```

Hayes-3c

Hitchcock: Aren't you saying that there exists something called REVERSE.

Hayes: NO! These are just a pair of first-order equations. If you give them to a theorem-prover that uses a depth-first, left-to-right, full substitution search strategy, it will do what LISP does with Hayes-3b. There is a uniform way of translating Hayes-3b into first-order logic. That is well-known.

Hitchcock: Does this mean I can express the equivalence of two LISP programs in FOL. (Hayes) No, you cannot. (Hitchcock) But I can translate both into FOL. (Hayes) You can translate the algorithms. But I am not saying you can translate the inscription of the algorithms. For that you would need a metatheory of FOL. I am not talking about proving properties of the programs. I am just talking about arranging that a deductive engine will do the algorithm.

I refer you to Keith Clarke or Bob Kowalski of Imperial College; or to David Warren of Edinburgh who implemented an evaluator for Hayes-3c-like programs in many interesting ways.

Goldstein: One might have lots of equivalent formulas but choose one versus another for reasons of perspicuity or whatever.

Hayes: Absolutely. All I wanted to say is that the distinction is merely one of perspicuity, or even less than that. It is not nearly as substantive a distinction as it is almost always made out to be. In particular, the idea that there is something wrong with first-order representational schemes, and that we must therefore move toward procedural ones, must really be laid to rest.

Control of Inferencing

Hayes: There is a problem, however. Nobody knows how to control the inferencing process. Nobody knows how to realize the appropriate strategic behaviors or search strategies from a bundle of first-order axioms given to some inference engine. I call that the control problem. Generally we reference Minsky 1903 [sic!], or something like that. The strategies in KRL and in Prolog are no good. There is a tradeoff between how much behavior should be specified and how much annotation you have to put on to obtain that behavior. This is a big research question. But, (a) the control problem does not have anything to do with representation languages and (b) it is always there. Whenever you can make inferences, you have the search control problem. And the particular syntactic sugar you use does not help solve it.

Rich: But syntactic sugar is important. A higher level representation may really help you think about your search problem, because looking at 5000 formulaes, you do not begin to know where to start.

Hayes: My own intuition is that argument has been overused. A lot of icing of cakes in the hope that the currants inside will get better. I think the procedural attachement idea is a copout. I think the way to go is the idea of metalevel descriptions of behaviour. I have no idea how to do it, but here are some people who have some interesting ideas: Randy Davis (MIT), Richard Weyhrauch (Stanford), Brian Smith (MIT).

SUMMARY

Deutsch: In summary, I asked the five presenters to respond to the sets of issues that I outlined at the beginning of the session. Their response to the purpose question is shown in Table 1.

The terms that were used are vague, but they do they do convey a sense of commonality of interests. The rest of their responses can be summarized as follows: The target population for all five notations was system specifiers and high level designers. These notations are all primarily descriptive, but, as several people pointed out, the procedural/descriptive distinct does not have much utiltiy. The formal foundation for all these notations was first order logic. This was somewhat surprising. Other formal foundations such as graphs, algebras, sets and mappings were also mentioned as useful for describing a notation. I could find no coherent way

Presenter	PURPOSE	DOMAIN	ARENA
Feather:	functional specification	behavior (data+ procedure)	arbitrary information handling systems
Hayes/ Reiter:	description (precise)	objects, relations, properties	all
Sowa:	model/ specification (precise and natural language	objects, relations, properties	database schemata
Carbonell:	description	inheritance mappings	hierarchical object/ attribute description systems
Rich:	description/ specification	programs, data structures	individual programs/ libraries

Table 1. Purposes of Notations

to relate the primitive elements of the various notations.

Finally, there were a number of different abstraction mechanisms mentioned:

inheritance (specialization/generalization)
transformation/mapping/multiple views
classification/instantiation
composition of wholes from parts
procedure formation (lambda abstraction)
specification -> implementation

Some of which are supported in each notation, but no notation supports them all. There was enough overlap, however, to justify the list.