Rowe: I am going to talk a little about data abstraction from a programming language viewpoint. I suspect that what I will say in some places will be every bit as controversial amongst programming language folks, as what Dennis said was amongst database folks.

Historically, programming languages evolve continually, from very low-level representations or descriptions of computations to higher-level descriptions. The idea is to express computations in a way that makes them easier to write, faster to debug, and make them survive change. All these marvelous buzzwords!

The content of this part of the presentation was the same as section 2 of Rowe's tutorial paper.

Now, let's consider the evolution in the description of data structures, or data abstractions in programming languages. Let's suppose we're writing a program that involves lists of integers and certain operations we want to apply to these lists of integers: create, take the first element of the list, take the tail (all but the head), insert at the front, and test for an empty list. How do I write a program which uses lists? What sort of notation do I use? In a language like Pascal this type of programming involves a very specific description of the list structure. It is some sort of linked list of records with two fields: "next" which is a pointer to the next list element, and "value" which stores the element value which is an integer. The program determines a lot of detail about the implementation, whereas the operations you want to think about are in terms of tails and heads.

Pascal is an example of a language that has type constructors and a set of primitive types out of which you can construct the base structures. Operations are defined using the other statements in the language. You can use procedures for operations, but in the past they didn't necessarily do that. Extensible languages tried to allow you to use the operators which are present in the language to express the operations on the objects. So, for example, you would use the same kind of low-level description of the entities, but now you'd use the operator "plus" to denote the "insert in front of" operation, so I could say "list plus one" if that was the way I wanted to express taking a list (one, two, three) and producing the list (one, one, two, three). Except in extensible language, the problem was that the operations on the representation were not restricted to abstract operarions. For example, you could assign the value 3 to the first node in the list, which is an operation of changing the first value. That is like deleting the first element and inserting the new element; however, that operation was not defined as invalid on that abstraction. It wasn't listed at the start. There was no protection provided by the extensible languages against invalid operations.

The data abstraction languages, starting with Simula, Clu, Alphard, and most languages designed since Simula, encapsulated or put the operations together with the declarations of the representation. For example, the record used for the list elements is declared together in one unit with the procedures used to implement the abstract operations. The unit is called a data abstraction. To use the list abstraction one would declare an instance x of the abstraction list of integers. To perform the head operation you say something like "head.x". The key point is that in data abstrction languages the operators are put together with the representation description which can be accessed only by the defined operations. There is an important distinction between x inside the body of the procedure "head" where it is an object of the type of the representation versus x declared outside the procedures as an object of type list.

Let me briefly summarize the evolution. First, Fortran and some of the earlier languages had a fixed set of data structures, e.g., arrays, vectors, complex numbers, etc. More complex structures could be built using the fixed set. Pascal, PL/1, Algol68 and some later languages introduced type constructors with which you can construct and define new types. These languages offered no way of binding the operations on the abstract object to the definition of those types. Following that, extensible languages offered type constructors and operator extensions but had no protection against doing one of the operations on an instance of another type. Lastly, the data abstraction languages encapsulate operators to provide the protection.

Data Abstraction: A Definition

A data abstraction is a specification of an implementation of an abstract entity and its operations. The operations are encapsulated so the only way to access or modify instances of the entity is by the abstract operations. "Specification of the implementation" means the real programming aspect of it. There are other approaches to abstraction that deal with specifying the semantics of abstractions and not the implementation at all.

Rich: Here you're using data abstraction as a single thing as opposed to a relationship between two things presented in the database tutorial. Isn't there a relationship view of abstractions in programming languages also?

Shaw: It seems to me that in programming languages there are two things that are being related by abstraction. One is the implementation; the other, which we call the specification, is the description that you may present to the user, who may be indifferent to, or even independent of, the implementation. In the relationship "x abstracts y," the specification is said to abstract the implementation.

Rich: It's confusing to me because what you call implementation and specification are both data abstractions in the sense that for both a set of operations and constraints on them are specified. Doesn't the representation function define the abstraction relationship?

Shaw: The representation function is the formal tool which relates a specification and an implementation.

Rich: So you have two levels of description; the abstraction is the relationship between them. Therefore as defined here, data abstraction is a specification technique.

Hayes: The relation of this notion of data abstraction to that given in the database tutorial seems rather remote. However, they both have something to do with the relation between the conceptual and implementation levels. The mapping between a clean representation and the nasty details.

McLeod: What you call the data abstraction here is called representation independence in databases. There is another aspect of abstraction within the conceptual schema itself.

Rowe: I will now discuss four data abstraction research issues:
1) the language constructs or facilities needed to define the abstractions;
2) ways of specifying the semantics or abstractions;
3) generic types or hierarchies, which relate to the work of John Smith;
4) problems having to do with type checking, and its role.

Design Issues for Language Constructs

First, some of the design issues for language conconstructs. In one of the early data abstraction languages, CLU, a data abstraction corresponded to a single abstract data type. There are circumstances where you would like an abstraction to define more than one type. The definition of that abstraction requires the ability to get at both representations of the internal object. As an example, consider a virtual terminal abstraction which shows multiple windows on a screen. I now want to define an operation, clip, which allows you to reach into one window, pull out something and put it in another place. It's very hard to define that abstraction in separate abstractions without a window abstraction with operations designed solely to implement clip. Those operations show through the implementation of window. So, in some circumstances it is more convenient to define that abstraction as having two types of things that are being dealt with. There are other examples where the data abstraction does define a single or zero types. It's just being used to collect together a bunch of procedures or subroutines in a subroutine library.

Leavenworth: What languages support what you're talking about?

Rowe: Modula is an example of a language that has a data abstraction mechanism, and Mesa, Ada and Euclid are examples of languages with data abstraction mechanism where you can export more than one type. In fact, most of the languages since CLU have provided some mechanism to do that.

Balzer: What about zero types? (Shaw) Ada has that. (Deutsch) So does Mesa.

Shaw: But the trick is whether you associate the name of the encapsulation directly with the type as Alphard did. If I were to redesign Alphard, I'd do it the other way.

Other Design Issues

Rowe: There are other issues in the design of the language constructs. One issue is the separation of the interface specification from the actual implementation. A specification is a definition of the interface of the abstraction including operations and the arguments to those procedures. It's much nicer to separate the specification from the implementation. First, it may be easier for people reading the abstraction not to have to wade through all the implementation detail. Second, you have a separate description of interface to the abstraction, which can be used for separate compilation.

Another design issue is, can abstractions be parameterized? If so, what kind of parameters do you support: compile time constants only, variables, or types. The choices you make have some rather important consequences for the implementation required to support them.

Lastly, does the language support facilities for applying constraints to objects which are passed as parameters? I intentionally worded it vaguely because there are two distinct cases I want to bring to your attention. One of them concerns type parameters to abstractions. You may want to constrain types acceptable as actual type parameters. For example, a type parameter of a sorted list abstraction must have some notion of ordering. The second example of constraints on parameters concerns protection. You may want to pass a particular list to a procedure but not let that procedure update the list.

Deutsch: I have two comments. One concerns the question of separate compilation. The question of compile time constants points up something that bedevils us constantly in the programming language area, namely, the confusion between language expressiveness and pragmatic issues having to do with the structure of compilers and language systems. It is very unfortunate that discussions of the two constantly get mixed up together. The second point concerns a notion of type I just want to throw out for people's consideration, which is: a type is a characterization of a set of objects which have *a priori* some structural or behavioral

properties in common. An instance of a type is simply an object which has those properties.

Rowe: Another big issue concerns the semantic application of data abstractions. The goal is to specify the semantics of the abstraction without actually giving its implementation. This work is analogous to work on defining the semantics of programming languages. The two principle approaches to semantic definition are the operational and definitional approaches. An example of the operational approach is the Vienna definition language, which has developed for defining programming language semantics. Examples of definitional approaches are Hoare axiomatic techniques and the algebraic relations technique.

[The algebraic relations technique is illustrated in section 3 of the tutorial paper.]

Now I'll tell you what my colors are. I'm very much on the pragmatic wing of programming languages. I'm very much concerned with how programming languages are designed and implemented and how programs are written. There has been a lot of work done on algebraic specifications, but to my knowledge, there has been relatively little use of those mechanisms in real programming projects. People writing ten, twenty, thirty thousand line programs are not taking time to do these kind of specifications. Basically the payoff associated with doing the specification has not been good enough to justify the time and effort.

Type Hierarchies and Generic Procedures

I want to highlight a couple of ways that there are hierarchies within type systems within languages. One hierarchy is formed by the instances of a parameterized abstraction. For example, queue(t) is generic; it has queue(integer) as a specific "subtype." It also has queue(queue (t)); that is a queue with a parameter of type queue(t), and of course the hierarchy can grow down there.

Weber: In what way is this a hierarchy?

Rowe: The abstraction queue is a generalization of the queue abstraction with a parameter which is in turn a generalization of the queue abstraction with a specific argument, for example, an integer argument.

Rowe: Another form of hierarchy is obtained with subrange types. This was introduced first in Pascal where 0..10 is a subrange of the type integer. The type 0..10 is a subset or a lower down on the hierarchy than integers. Similarly, you can define subranges of other scalar or enumerated types. For example, the enumerated type orange, red, blue, green, grey, contains the subrange blue..grey which has the elements blue, green, and grey. Another form of hierarchy has to do with union types, where a type can be a union of the types, e.g., T as a union of integer, real, and complex. At a given time, an object of T is one of the unioned types. A variable x declared to be of type T would be either an integer, a real, a complex, or unknown if the type has not been determined. In some languages you can change the type of that variable, x, to be of one of the other types defined in this union. So it might be an integer for a while, and then it might turn into a real. Union types are useful when the computation involved is the same except for the types that are being used.

Rich: In what language can you change the type as you go? (Rowe) Algol68. (Deutsch) Any language that doesn't have type declarations. (Shaw) Pascal gives you variant records. (Meyer) Simula also.

Rowe: A generic procedure is one that operates on objects of different types. For example, the operation sum could operate on different types which in the program are of type parameter. An important point here is that the plus operation comes from the actual parameter type. The plus operation differs depending on the particular parameter. In generic procedure languages you must be able to check that the necessary operations exist. If a needed operation doesn't exist then you want the language to flag that as an error.

Type Checking

The last thing I want to talk about is type checking. The goal here is to look at the types of objects which are being operated on and see whether those operations are valid on those objects. If the operands of the operations are required to be of certain types, we want to be sure that the actual operands are of those types. There are two fundamental approaches, name equivalence and structural equivalence. Neither works perfectly.

The content of this part of the presentation is the same as section 8 of Rowe's tutorial paper.

Let me briefly compare the two approaches. Name equivalence is easy and efficient to implement because you simply look at the names or just check whether two pointers are the same. However, you can get into problems type checking library functions. For example, a programmer calls a library function, sum, with a parameter which is of name type t. Sum wants to work on the representation of its parameter t using strict name equivalence, the description of sum could never be type equivalent to the name type created by a user. You need some additional mechanism to unseal the type. Structural mechanisms have really complex rules; there are quite elaborate descriptions of when two types are structurally the same or not.

Codd: Don't you need both approaches. And some notation that distinguishes the two?

Shaw: I basically object to characterizing type checking this way. The notion of type equivalence and type safeness is a red herring; it leads us down a long and very complicated track. The real issue in type checking is if you have a definition of a procedure which includes some formal parameters you want to determine for a given call to that procedure if the actual parameters are acceptable. The issue has to do not with whether two variables have the same type, but whether the variables or the expressions used for actual parameters are acceptable to the formal type descrip-

tion. Type equivalence of two variables is a red herring in trying to answer that question.

Rowe: I agree with you. That is the more precise description of the issue; however, type equivalence does illustrate some of the approaches to doing that. I'm trying to illustrate two extremes without looking at the more complex cases. One extreme is the name equivalence approach in which objects have to be of exactly the same names type period. In the structural approach, two separately defined abstractions are equivalent if they have the same number and names of procedures in the same order, with the procedures and representations equivalent.

Shaw: The direct answer to Ted's question is that the languages that implement one or the other of these do not provide for tagging one or the other.

Rowe: That's right, and I'm not sure that the answer is so simple as putting both in.

Hendrix: You can get the name from the structural one by using a tag in the structure.

Rich: Type equivalence is not a red herring. It seems to me, from the standpoint of this workshop, that equality between objects, whatever their description, is very much an issue to be discussed. Type checking is a purely technical thing within the realm of programming languages. We may have a corresponding thing which remains to be discovered in databases and AI, but I think that type equivalence is a central issue.

Rowe: Besides the complexity of the rules there are other problems with the structural approach. The most general case is difficult to implement since it requires complex algorithms that take a lot of space and time. Due to the additional complexity and cost, it may not be worthwhile to implement those algorithms. Another issue with the structural approach concerns protection; that is, preventing other people from operating on a type. The protection aspect of typing is not there since in structural type checking you reach into the descriptions to check for representation equivalence.

Shaw: Would you clarify your example for structural equivalence. Is it possible under your definition to write a procedure that accepts arrays that have 10 integers and that does not care about the index set?

Rowe: It depends upon what the definition of structural equivalence in the specific language. Some languages will require the type of the index set to be the same type or structurally equivalent.

Deutsch: I think Mary's example is a wonderful one. It indicates that what you call name and structural are only two very small focuses of attention on what it means to be a type. If you take the notion of a type as simply being a partial description of a set of objects, some common attributes of a set of objects, then you ask for a given language, is there a way of stating in that language the characteristic "has n elements" as opposed to the characteristic "is indexed by the subrange 1 to 10." It seems to me that that's real issue we're dealing with here. I think this is going to come up again in other contexts.