Rowe: Data typing is a technique that researchers in the various areas use to solve a variety of problems. This session focuses on how types are used in a particular domain to solve particular kinds of problems. The long term goal is to answer the question, what is a type? To begin we will have presentations by representatives from the three areas, AI, DB and PL, on how they see types being used. Mary Shaw will begin and will be followed by Ira Goldstein and Ted Codd.

THE NOTION OF TYPES IN PROGRAMMING LANGUAGES

Shaw: I will begin by stating some definitions. These definitions reflect the way I use words like abstraction, model and type. First, I don't agree that abstraction is another word for good. Abstraction is sometimes used as a noun and sometimes as a verb, if it is used as a noun abstraction is a description of a system in which some details are emphasized and other details are suppressed. A good abstraction is an abstraction in which appropriate details are emphasized; thus, goodness depends on context. Abstraction is relative to the use for which the descriptions are to be used.

I don't use the word model very much in a technical sense. I have been trying for a couple of days to understand why there is this distinction between a model and an abstraction. When I say model or abstraction in a general sense then I am not attempting to make a distinction.

The most important thing about types is that types are not God given and type is not a central part of the universe. More than anything else, the notion of type is a means of organizing information about programs. It forces the program author to deal with certain issues. It is a means of summarizing information about values common to a subset of the program's variables. It is a means of expressing the kinds of operations that may be permitted on those values. The motivation for type arises from a need for discipline in programming. It arises from the need to maintain the integrity of data, and in recent times it is used to reduce life-cycle costs of programs.

Abstract Data Type Definition

An abstract data type is a user-defined type. It has two parts, a specification and the implementation. The specification contains information that may be used by a client, defines the interface to be used by other parts of the program, and guarantees that variables of this type will have certain properties. This is coupled with a protection mechanism that insures that no part of the program destroys the integrity of the data values.

In PL there is a strong flavor of protection, encapsulation and scope rules. That is, the ability to refer to specific components of the implementation may be restricted. Only privileged portions of the program inside the definition of the data type itself are allowed to refer to and manipulate these components. The specification also describes all of the operations.

Thatcher: Could you give a working definition of the noun "type." You don't mean it to be a means of associating properties.

Shaw: As used in a program, type is an attribute of a variable, that associates with a variable certain information about how the variable can be used. You may also think of a type as a generation mechanism, i.e., a template used to create variables for use in a program.

Buneman: Does this mean that in variable-free PL the concept of type is irrelevant?

Shaw: The way I have presented the concept of type is directed to variables.

Balzer: Don't you think it is the other way round, that type is the primitive idea and variable is an object of some type? In some sense attaching type to a variable is derivative from the notion of type as encapsulation. Presumably a variable is a variable because it can refer to different objects at different times. Now, is the type associated with those objects or is it associated with the variable that appears in the program?

Shaw: The type may be associated with the value, and type information may be associated with the variable itself. Commonly the variable/type relationship is one to one, because variables can only refer to values of one type. There may be variables that refer to values of more than one type. In languages such as APL, type is associated with the values a variable refers to.

Balzer: What I am saying is that types must be associated with values or objects. You may also associate type with a variable which is then restricted to a certain set of values.

Shaw: I would say "type information" is associated with a variable.

Deutsch: The question of whether a type is associated with variables or a type is associated with values is resolved in many different ways in many different programming languages. For example, in the programming language RUSSEL, types are associated only with variables and not with values at all.

Did you deliberately say that the abstract type contains both the specification and the implementation? Isn't an abstract type composed of just the specification?

Shaw: I was referring to the entire activity that I call abstract data type definition. With respect to the notion of abstract type you are quite right. The part that I want the client to refer to and understand when he is using the type is only the specification. In the process of program design and development one may define a specification without necessarily providing an implementation.

Thatcher: Isn't a type a set of values and its opearations? (Shaw) Yes.

Hardgrave: Presumably you can have operations on types?

Shaw: There are systems in which type is treated as a type, and systems where type is not treated as a type. Without type as a type there are certain things you can't do.

The following is a historical remark. As I looked at the last ten or fifteen years, it seems to me that there has been a growth of abstraction techniques of various kinds. It seems that the areas of growth are strongly related to the subjects which we have at the time been able to specify formally.

Rich: I have trouble with "abstraction" as a noun. It seems to me that nothing is intrinsicly an abstraction. For example, people talked about sets as being abstractions. I would prefer a more technical definition of what we are doing. I see abstraction as a relationship between two specifications, i.e., formal descriptions.

Shaw: I think that is consistent. The purpose of an abstraction is to make the problem more manageable.

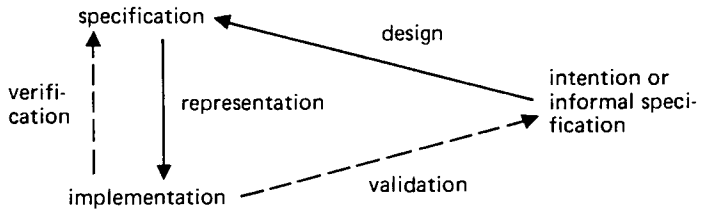
Rich: In AI that paradigm has been called "problem solving by degugging almost right plans." In order to get a simple problem description which can be solved easily and for which the solution is recognizable you may make an abstraction that might actually ignore details of the problem. When the ignored details are reintroduced, you may find that you have to debug your first solution to accommodate the new details.

How Models are Constructed and Used

Shaw: I want to talk about modelling in science. In discussions here, I discovered that different people are "modelling" with different degrees of restrictions. I am using it in a general sense. My view of experimental science is that we design models as aids to understanding pieces of reality. First, the inputs affecting the behaviour of that piece of reality must be identified. This is not always easy. Given the relevant inputs the next step of building a model is to formulate a description of how the inputs relate to each other and how reality can be described in terms of those inputs. The description (model) can then be used to produce outputs which might be predictions.

In the design of the model you examine reality and attempt to identify facets that are pertinent to the question you are trying to answer. Modelling is relative. The notion of what is pertinent to the model is very important. In science, you then attempt to validate the model to discover discrepancies. You refine the model until at some point you decide that it is good enough for the intended purpose. That's my view of modelling in the large.

Consider the following diagram in applying the above general structure to the problem of formalizing a model using abstract data types in programming languages. We start with a set of intentions



or informal specifications instead of a piece of reality. In the ADT world, that which we call the formal specifications corresponds to the output of the model and the input to the model is the implementation of the ADT. The description of the model shows the consistency of the implementation (inputs) with the specifications (outputs). This is the process we call verification. We can only perform verification via formal reasoning for statements written in formal mathematics or in PLs that are precisely described. The verification process only insures the consistency of the code and the formal assertions. It does not insure that either matches the informal intentions that you had in mind.

Rich: Can't you replace validation with testing in the scientific paradigm?

Shaw: Yes, the modellor would begin by designing his data type and then writing down some specifications for it. He would then try out his model against his intentions and discover that it does not quite fit. Here he proceeds as he would in the scientific paradigm when he discovers his model doesn't quite match what he had in mind.

Balzer: I am surprised that the input to output mapping goes from implementation to specification, rather than the other way around. Additionally, I have trouble understanding the correspondence between the implementation and the model inputs, on the one hand, and the correspondence between the specification and the model outputs on the other.

Rich: Shouldn't the inputs and outputs be reversed since the inputs to the model come from reality and the outputs should be validated against it. The input to your model corresponds to the specifications which you get in trying to capture the informal intentions.

Shaw: I understand what you propose, but I am thinking from an execution standpoint. I am

really talking about the process of abstracting a formal specification from the implementation.

Rowe: Testing the implementation against reality may be more effective than taking the time to do a specification. The choice of what is in the specification is very critical in determining how much help you get from the specification. Some people confuse the use of formal notation with the accuracy or certainty of the specification. The use of formal notation does not prevent the omission of important details that get noticed only when an implementation is attempted.

Deutsch: People have different opinions of the value of formal specifications and proofs. Let's not debate that issue here.

Problems with Using Abstract Data Types

Shaw: I have two lists of problems that arise in using ADTs. The first list has limitations of the methodology, the other list consists of problems in actually practicing the methodology.

There are three main limitations of the methodology. First, the concept has not yet been validated in practice. I don't know of a system in which ADTs have really been used in the design, implementation and verification of the system. Second, the specifications of the data types we write down are purely functional. For real programs there are many other properties about programs that matter, performance for example. The other limitation is that third, although ADTs usefully organize information in a program, they are not the only way.

Problems with Practicing the Methodology

Shaw: The main problems with practicing the ADT methodologies in PLs are: In most cases, only one type may be defined in a single module. Types must fit a strict hierarchy. (Rich) In a strict hierarchy, an object cannot belong to two dissimilar types. (Shaw) Yes, but I wouldn't want to do that. Some people do.

Shaw: There is a problem defining when multiple specifications for a single type are sufficiently alike.

Thatcher: If you have a working definition of type then you know what sufficiently alike means. For example, the two specifications denote the same set of values and they define the same operations upon them.

Shaw: I think that definition is too primitive. For a given application, I may have two specifications that might have slightly different value sets and operations. The only operations that are required by the application, however, are those in the intersection of the two specifications. Thus, two specifications are different but they are sufficiently alike for that specific purpose.

Thatcher: Let me illustrate my point with an example from mathematics. You can define groups in terms of the division operation and certain axioms, or you can define groups in terms of multiplication, inverse and identity operations. Both definitions give rise to the same theory of groups, but with totally different presentations. I would suggest that what you really want as the notion of sufficiently alike is that the two specifications give rise to the same theory.

Rich: That may be too strong. You need a weaker notion of alikeness for some applications that don't behave exactly the same way.

Weber: Why would you say these contained the same type.

Shaw: The motivation for raising this issue is that in developing a program, it might be beneficial (from the performance standpoint) in changing the implementation of one of your types. It may also be the case that the specification for the alternative representation is slightly different than the specification being replaced. The differences, however, do not matter to the application in which the type is used.

Some of the other constraints on practicing the methodology are: the type checking rules of the PL you are using, the kinds of information that can be represented in the specification (for example, performance as well as functional specifications) and implementation issues such as the existence of separate compilation, the ability of the user to define and manage his own storage of objects and when (e.g., compile time) is type checking performed.

Filling the Operation Gap

Christian: The main use of ADTs is to fill the gap between a set of values and operations that a user of a system would like to have and the set of values and operations provided by a machine. The user would fill in the conceptual gap with a hierarchy, both external and internal views of the ADTs are needed.

An ADT is a set of values and a set of operations. The external view (the specification) of a data type must define the set of values. In ALPHARD the user has to define the set of values explicitly in terms of mathematical entities (sets, vectors, functions). In the algebraic approach the set of values is defined implicitly; for example, as equivalence classes generated over an algebra with a relevant signature. Operations are characterized by state transition relations. State transition relations are given implicit definitions in terms of conditions on the possible system states before and after the operations. The conditions are expressed by predicates. The precondition predicate specifies the domain of the state transition relation. If the precondition is true, there exists an abstract state which is the desired state transition. The postcondition is a predicate which is true if and only if a given post state is paired with some prestate in the state transition relation.

The internal view of an ADT is where the representation of the abstract states is attempted in terms of the states that are provided at the concrete representation level. A concrete state

transition relation (in contrast to the abstract transition relation defined above) can be defined in terms of the concrete states related by the composition of concrete primitives in the operation implementation. Each of the concrete primitives has its own implicitly defined (concrete) state transition relation.

The external and internal views are related by the abstraction function. This gives a feeling for the gap between the two views. The abstraction function is constrained to map these concrete states meeting the implementation invariants into abstract states so that initial concrete states go to initial abstract states (i.e., states that satisfy the relevant precondition). If two states are within the derived concrete state transition relation then their abstract maps must be in the abstract state transition relation.

Implicit versus Explicit Definitions

Wedekind: You are viewing computer science as a sub-branch of mathematics. This creates a serious problem in that you introduce your objects implicitly. This is well accepted classically. Even Hilbert said: I don't care what my objects are. He does not care about the real world, he just intends to prove that his system is without any contradiction. We should not use implicit definitions in computer science. We will never be able to fully capture an application world in an implicit definition.

Rich: Could you give examples of explicit and implicit definitions.

Wedekind: In an explicit definition I just replace one character string by another. For example I define a personal number as equivalent to a name and an address.

The axioms of geometry constitute an implicit definition. The notions of point and plane are implicitly defined by axioms.

Rich: How would you explicitly define a plane.

Wedekind: You come up with another geometry. Then I have to talk about a point, what a point is and then build up a structure.

Rowe: Ira Goldstein will now talk about types from an AI viewpoint.

THE NOTION OF TYPE IN AI

Goldstein: In the work I am doing the distinction between PL, DB and AI concerns doesn't really arise. The following example of a "space war" implementation illustrates that from the PL standpoint, how would you implement this particular simulation? In a functional language like LISP, one would have a collection of functions to move and accelerate. In a simulation language like SIMULA you might create a data type for space ships and associate those data types with the particular behaviour that you want them to have.

I am working with a language called SMALLTALK. In that world one would implement this simulation in terms of an ADT which in SMALLTALK is called a class. The actual spaceships are instances of the class (in SMALLTALK terminology). You might have a particular instance called "Enterprise" that has a particular state, a certain position, a certain velocity, acceleration, etc. Each is an instance of the class spaceship, where spaceship names the state variables and defines the method.

Weber: Can you explain what method means?

Goldstein: A method is a procedure; for example, one that works out a new position, velocity, etc.

Rich: Mary Shaw talked about ADT as having a set of operations, would the functions here (velocity, position, acceleration) correspond to the operations in the ADT. (Goldstein) Yes.

Goldstein: Every ADT may have a whole bunch of methods that it supplies. The internal implementations may define some methods in terms of other methods and you can probably guess from knowledge of physics that the primitive functions for position, velocity, acceleration are not unrelated.

A Database Viewpoint

Consider the issues that arise from a database viewpoint. My goal here is to show how three different viewpoints can be applied to the same problem. Suppose you want to simulate the spaceships flying around in space. You might like to treat that world as a database of which you could ask questions like "list all spaceships in a particular sector." In the SMALLTALK world, achieving that capability requires other kinds of methods that inspect the class.

At this point it is no longer desirable that a class implement all the behaviour of a data type, but it's useful to have the word class because here we get some of the behaviour of the data model. The behaviour of the class "spaceship" includes a function for deciding whether the position lies inside a certain sector. You can think of various other kinds of support for different kinds of queries for this world. The class has to organize its instances if there are large numbers of them. In general, an ADT is not organized in a database fashion with indices to its instances. A data model is so organized, however, and to have the spaceship respond to queries we have to introduce various kinds of organizational principles. In the current SMALLTALK, the number of instances is small and so we typically do linear searches.

Balzer: I think it is important to point out that in the SMALLTALK world, there is no generic system information language. Everything in SMALLTALK is locally defined, and the same thing is true of an ADT. If there is system-wide consistency it is because people have agreed by convention that it should be so.

Goldstein: Since the SMALLTALK world is hierarchic, if you have a data type called object then behaviour that is supplied with objects bear on all objects in the system provided you have not overridden it by a subclass. AI people worry about recognition problems. In the traditional database world when

you have a tuple, the types of its domains are well defined and are known immediately from the type description. The same is true in the PL world. The recognition problem consists of attempting to identify the type of an object from evidence that has been accumulated about the object. Other issues that arise are the storage of default information, the representation of constraints on data operations and the problem of dynamic reclassification. In the latter case, you might get enough evidence to assign another type to an object. You may also have a nested set of descriptions. The following mini-model illustrates the relationships of the issues raised above. You can be in the plane of the actual computing process. In this plane you think about typical issues related to PLs. You can be above that plane looking down at it and ask questions like "Do I have an object in that plane that participates in certain relationships?" That's a kind of data base viewpoint. I put the AI viewpoint outside the system as a whole, since it is concerned with recognition of types in and constraints on that system.

THE NOTION OF TYPE IN DATABASES

Codd: I am very uncomfortable talking about real world objects or reality. I use these terms because I feel it is necessary to communicate with people who use these terms, but I feel that representation, specification, implementation and abstraction are really all fundamentally relationships between objects. In English, unfortunately, the word that is used to name a binary relation is often used to name the range of that binary relation as well. For example, a representation is a relation and the target of that representation relation is also called a representation. We should really distinguish between representation relation, representation source, and representation target. The point of this statement is that there are a lot of fuzzy concepts using fuzzy words and throughout this whole workshop we are trying to be more and more precise.

I want to mention how type came into the database world. We need some intuitive concepts on which we can base any formal definition of type. Associated with the notion of relation is the notion of a relation schema that consists of the name of the relation together with the names of the attributes that occur in the relation and the domains they draw their values from.

Associated with the relational model is the idea that some combination of the columns or attributes has an identification property; that is, the values associated with that combination of attributes will distinguish every two rows of the relation. If no attribute in the combination you have chosen is superfluous, then you call that combination a candidate key. There can be more than one candidate key in a given relation. One is by convention chosen to be the primary key. Associated with that choice is the intuitive idea that the value of the primary key somehow represents or distinguishes certain real world objects from each other.

So there was a notion of type in the following sense, both the representation source and the representation target had a type associated with them. This notion of type was also being used to constrain operations like joins. The system would either prohibit (or, at the very least, alert the user to problems if he tried) a join of two relations over two attributes with different domains.

The next notion is a referential notion. Let's suppose we have several relations and they have attributes over the same domains. Let's look at an occurrence of a specific value in an arbitrarily chosen column. What does that specific value mean? Obviously, we have more than a value; we have a value which is tagged with the name of the domain that column is defined over. This value cross references every other occurrence in the database of the same value that is similarly tagged. The cross references cannot be replaced by a simple pointer, because of the multiplicity of references or connections.

Initially there wasn't a good notion of subtype in databases. The introduction of it must be credited to John and Dianne Smith with their data abstraction papers. I think a notion of type is only as good as the notion of subtype that is associated with it. Subtypes have been explored by a lot of people previously in the AI world and some in the PL world. If you have a database about people and then you want to record specialized information about certain subclasses of people, then you get into these subtype considerations. It is important for the database system that it know which things are subtypes of which. There are questions to be asked about the notion of relation type. A relation has a compound domain composed of the domains of the columns in the relation. This compound domain can be thought of as the type of the relation. Is this the same thing as a tuple or record type? From the standpoint that a type consists of objects and operations, they are not the same because the operations on relations are not necessarily the same set as the operations of tuples.

Consider two relations each having only two columns, defined over the same domains. Are these relations of the same type? Are they the same type if the applicable operations are also the same?

One might take the view that there are other reasons to say that things are of different types, one might say from the standpoint of sets and operations they are of the same type. There is the question of whether there should be a nominal distinction between types, or just a structural distinction. There is a need for both nominal and structural equivalence.

Reiter: In an earlier session McLeod suggested that quantification of relations can be a type. You were talking about typing a relation. What I understood from McLeod's talk was that, on occasion, a relation itself can be a type not that a relation is typed.

McLeod: What I meant was that you can associate a type with a relation, and that the tuples are the instances of the type.

Weber: If a relation is of a certain type, are all tuples in the relation instances of the type?

Brodie: The tuple can have a type, and the relation can have a type, which is "a set of tuples of the tuple type."

Schmidt: Rather than expressing the relationship between tuple instances and the relation they reside in as a relationship between the associated types, it is better to use the value sets associated with those types. The value set associated with a relation type is the power set of the value set associated with the typle type (restricted by the unique key condition). This means the relation may range from being totally empty to containing all possible non-redundant tuples.

Transactions

Wedekind: Isn't the notion of a transaction much closer to what is here being called a type?

Codd: I think transaction corresponds to an encapsulation in some sense. But it has some additional properties that are not discussed in the PL world. It is an all or nothing thing; that is the database may be inconsistent if the transaction doesn't complete. That notion seems to be somehow missing in the PL's. Transactions also control concurrent access to shared data and automatically handle the acquisition and release of locks. This kind of high level synchronization seems to be missing from PLs.

Shaw: As part of the definition of an ADT, an invariant relation is stated on the representation. This invariant relation specifies the integrity requirements on the representation. The invariant relation is expected to hold when an operation is begun, and it is expected to be restored by the time the operation completes. The invariant relation may be violated during the execution of the operation. In that sense the atomicity of the transaction does have an analog in PLs.

The semantics of sharing are not well understood. The closest claim I can make to a sharing mechanism in PLs is a "monitor" in which many of its properties are similar to ADTs but in addition there is a synchronization mechanism.

Codd: That is completely inadequate for database purposes.

Hitchcock: There are other areas where databases differ from PL. For example, it is not exceptional if something goes wrong in the middle of a transaction. A good database system will actually back out what has been done to the database and restore it to a previous state. With a PL you tend to assume that those atomic actions are going to take place. It's up to the programmer to recover if they do go wrong.

Zilles: I disagree slightly with both of the last two comments. First, from the PL viewpoint, the literature on the readers and the writers problem discusses many ways of realizing atomic actions in concurrent processes. The real problem is that there are some things that are atomic at one level, but are composed of more primitive actions when viewed at a lower level. That is certainly what happens in a database system. A transaction as a whole is atomic, but it is also made up of a collection of other actions which each by themselves have the status of atomicity.

Crash Recovery and Exception Handling

Zilles: My second point is that the existence of exception handling in PLs was motivated by the need to restore the integrity of the data that was in use when the exceptional condition was discovered. The distinction between a transaction failure and an exceptional condition handler is that the exception handler need not undo all the previous actions but may, instead, attempt to restore an integral state by any means available to it. This form of recovery may result in the loss of information but it can be significantly more efficient.

Rowe: I would like to emphasize something Steve said. Transactions are designed to expect a crash occurring for external reasons over which the program has no control. One wants to survive such situations. Transactions help to guarantee that, regardless of whether the head falls on the disk you should be able to back up to the previous integral state. Exception handling mechanisms in PLs have some of the same flavor (in the sense that there are excpetions that can be generated) as the result of a hardware failure or the occurrence of an unexpected condition. From my observation of the two communities, it looks like database systems have a much stronger commitment to data preservation than do PL processors. This commitment causes them to do things differently than the PL community.

Balzer: Motivated by the need for reliability and robustness of systems, Brian Randall has adapted the notion of transaction oriented processing for PLs. From a specification standpoint, his approach makes the whole system much cleaner. This is an example of a DB notion being adapted for PLs despite severe implementation overhead.

Christian: I have three comments. First, concerning your observation that in the database they recover from crashes without looking at the precondition of this crash. In operating systems people work much closer to the machine. They can identify the preconditions of the exceptions and adjust their recovery procedures appropriately. They distinguish exceptions and failures. The difference between databases and operating systems is related to the conceptual distance between the physical machine and the DB. A DB interface is simplified. Using operating systems and PLs you distinguish more events.

Sibley: In the DB area we look at data as the resource and not the program. We are trying to preserve data integrity. There is a lot of money rolling on the fact that the data is correct. It is much easier to back up a database, find the

error and redo transactions.

Christian: The second comment is that there is atomicity with respect to both synchronization and exceptional occurrences (e.g., crashes). The relationship between those two notions is not very clear. They are not equivalent. The relationship is being studied in Newcastle. My opinion is that one can rely on the other.

The third point concerns exception handling and automatic backup in PLs. Exception handling mechanisms have been proposed for PLs. An exception occurs when the internal state does not satisfy the representation invariant, e.g., some modified variables are inconsistent with respect to the representation function associated with the abstraction. It is only necessary to recover these variables by relying on semantic knowledge about the implementation of the abstraction. Backward recovery mechanisms proposed by Newcastle and for database transactions do not use any semantic knowledge about the abstraction. One could combine the two approaches to achieve one which falls between using an exact estimate of what has been damaged and a very rough estimate of what has been damaged.

Hitchcock: In the Newcastle work there are two kinds of error recovery, backward and forward. This distinction exists in databases as well. Backward error recovery involves backing out to a state that you know is ok, but there may be some things you can't back out, like letters you have sent, etc. To resolve these discrepancies, you must use forward error recovery to patch up what's happened and come again to a consistent state. For example, if a bank messes up one of your checks, they don't back up their data base to a state of three weeks ago; they put in a correcting entry.

King: A central theme in certain kinds of AI programming involves setting up certain expectations and generating a hypothesis from them. If your expectations are violated, you want to have procedures that will set up a new hypothesis and decreasethe confidence in the old one. So the exception handling mechanism is used for reasoning rather than just an error reporting mechanism.

Zilles: It is important to consider the context in which recovery is to occur. Not all DB systems need to be backed out to achieve recovery. Most airline reservation databases have the nice property that after a flight occurs all related information is of little interest. You can clean up many airline databases by destroying all old records. Hence, many such systems do not need to be able to back out transactions. They are willing to live with dirty data. Recovery then depends on the environment. The application environment ought to be specified as part of your modelling activity.

SHORT DEFINITIONS OF TYPES

Rowe: J. Thatcher, P. Deutsch, R. Meyer, S. Zilles, P. Hayes and H. Mayr will give short presentations on their view of what a type is.

Thatcher: I wish that a data type were a family of sets together with a collection of operations or procedures on those sets. I wish that an abstract data type were the isomorphism class of a data type. So an ADT would be a concept which eliminates all representation and implementation details. Finally, I wish that a data type specification would have as its denotation an ADT. The reason I say "I wish" is because then the mathematics would be much clearer.

Cattell: Your definition of a data type is very like the definitions of abstraction in which an abstraction is a family of types together with a family of operations.

Rich: Would everybody please give the purpose of the definition they are giving?

Thatcher: My definition allows precision.

Rich: That is not enough, precision in itself is not a goal.

Thatcher: I need a definition that is precise enough to be workable. For example, parameterized types are much more complicated than people recognize. Unless I have a good, workable mathematical definition of type the notion of parameterized type is hopeless.

Weber: Another purpose for a precise definition is the verification of correctness.

A Definition of Type

Deutsch: I am not concerned with type as a formal mathematical entity. I am concerned with type as a notion that is useful to PL, AI and DB system designers. For that reason, I am asking you to accept a notion of type which has very many fuzzy words in it. I ask you to accept this definition as a way to think about of what you as a designer of types are going to design.

A type is a precise characterization of structural or behavioral properties which a collection of objects (actual or potential) all share. An instance of a type is an object which has the properties characteristic of the type.

Notice that there are some important undefined terms, such as property and object. The design of a type system consists of choosing particular meanings for the object and the properties and the notations. There are lots of things this definition does not address. There is the question of subtype. To my way of thinking, the subtype hierarchy is simply created by invocations between these characterizations.

There is the question of nominal versus structural equality of types. You simply choose whether you consider the nominal properties to be among the distinguishable properties.

There is the possibility of multiply typed objects, which arise more in AI than in other circumstances. This is another choice that you can make, that is to what extent you consider your type system to be a power set of some sets of elementary types.

The utility of a type system is basically that it allows you to partition the universe of objects in useful ways. Usefulness means that these distinctions reflect distinctions that are important to the uses of the system, the designer of the system or both.

In order for a type system to interact usefully with a PL or a DB system, the properties of your type system have to be in some relation to the operators and relationships in the rest of the system. For example, the notion of integer is probably inherent to the PL and must be covered by your type system. This may lead you to distinguishing between types of values and types of variables; that is, the type properties that are inherent in objects and those denoted in program fragments or descriptions.

Finally, "objects" should not be considered limited to data. You can also define a type system for procedures and you can make a type system for relationships.

Thatcher: The algebraic definition that I gave is consistent with what you have said. (Deutsch) Absolutely.

Thatcher: You said a type is a precise characterization. You also said you are interested in developing a type system, but you are avoiding the question of what a type is. A "precise characterization" may be a specification. The process of specifying or precisely characterizing is different from what you specify or what you characterize.

Deutsch: I will think about that, I am not sure that I believe in that distinction.

J. Smith: You said that a type system partitions the objects in your data space. Do you really mean partition? (Deutsch) The partitions need not be disjoint.

Meyer: Three ways of looking at programming objects are physical, constructive and functional. The physical viewpoint corresponds to programming in assembler or Fortran. In the constructive viewpoint objects are created from a set of logic entities by operators. This approach was first taken in Algol 60 for the construction of algorithms by combining a small set of basic constructs. Hoare applied the same approach to data types. In this approach new objects are created by applying operators to already created objects. The functional approach is much closer to the human view. This approach starts with a language that has an encapsulation mechanism, such as SIMULA and its successors or even higher level abstraction mechanisms such as ADTs or those of more recent specification languages such as CLEAR.

There are several ways to define "type" depending on the level you are at. The basic problem in programming (especially with ADTs) is to go from the functional or specification level, to the constructive or logical level, i.e., from an implicit to an explicit definition of the type.

Type Systems

Zilles: I want to look at types from an operational viewpoint. I agree with Deutsch's definition of type, but it becomes very interesting if you are trying to embed typing into an actual system. The system I am trying to embed it into is a PL. One issue here is insuring that when I apply a function to a set of arguments that the realization of the function will know what to do with each argument. Consider the function "pop" defined over sets of stacks that include the empty stack. That particular operation understands what to do with the empty stack. Among other things the function may raise an exception. But, at least the empty stack is accepted.

In this context, I came to the realization that the focus of typing should not be on types but rather the system of types; that is, a families of sets for which a given set of operations exists. In this case, the properties that are precisely characterized are the signatures of the applicable operations.

Given this view of types in a PL, there are two separate and important issues concerning the type hierarchy. The first is the definition of generalization, for which it is not the set of values that is important; it is the existence of certain operations on (properties of) those values. Consider writing a sort routine to work on arrays of type T. All that is needed is that the type T have an operation "less than or equal (LT)" which takes two arguments of type T and produces a Boolean result. Any type, such as integer, real or character is in the generalization having an LT operation and homogeneous arrays of elements of the generalization set are suitable inputs to sort.

Generalization has a separate notion of set union in which any sets may be combined into the union set. There may be no operations defined on all elements of the union. Operations defined as the union, such as addition over the union of integers and reals, are distinct from the equivalent operations on the primitive types in the union.* I would like to comment on your sort routine. For procedures, taking conventional kinds of arguments, programming practice provides type checking mechanisms using formal parameters. We do not yet have type checking mechanisms needed for your sort arguments. We don't know how to specify formal parameters for such cases. *Thatcher's remark.

Zilles: What I mean is that in the calling environment there is a type system which is a collection of sets and functions on those sets. At the point of call, a particular array will be chosen as the argument of sort. For that array type there must be an LT function.

Shaw: In my experience the set of type discriminations you really want to make there is very, very rich.

Zilles: I am talking only about the notions of type that can be evaluated at compile time, not runtime notions such as dealing with the empty stack. Some people consider these runtime issues as belonging to the definition of type.

Balzer: This discussion of types is very good since unlike the other definitions it deals with how types are used. The impact of saying that X is of some type depends on its use. The key to that analysis is the notion of characterization. The type, in this sense, is nothing more than a shorthand, for writing out some long expressions that say, I am interested in objects that have this set of properties. These properties can then be assumed for all uses of that object.

Hendrix: That does not always work. For example in an AI system one may know nothing about an object except its type. Consider a system with three types called "people," "men" and "women" with no proposition to distinguish them. I just know that these are subtypes. I may be able to determine that John is an instance of men and not an instance of woman.

Balzer: If you know nothing except its type, the type is the shorthand for the set of properties you know about the object.

Hendrix: In some cases you cannot state a single property.

Two Notions of Type

Hayes: As a definition of type I propose that a type is a subset of the universe we are considering. The notion of subtype is one of a set relationship. Types are introduced in AI systems to take certain properties of things out of the general purpose inference machinery and to utilize them more efficiently. For example, there is a lot of discussion and literature in AI about ISA hierarchies.

There is a different notion where you have objects that are constructed somehow. Then, the type has to do with the mode of construction. This is the kind of type you have in PLs. The motivation there (given by the last speaker) is that the operations that work on them have to know how they are constructed. That is a very difficult motivation from improving the efficiency of inference. For example, if you have an axiomatic theory of numbers, then there is no reason why integers can't be a subset of reals. If, however, your integers and reals are constructed, as in ALGOL or axiomatic set theory, they are different kinds of entities. There is a very different notion of a type hierarchy from these two different motivations.

I don't think that the definition of type is all that important. What matters is the structure that a set of types has, a data type system if you like. I can think of three important cases: disjoint types, a tree hierarchy and a lattice; there may be more.

Even more important are the functions (or constructors in PLs). How complicated can they get? The simplest case is that every function has its domain and range fixed. A more interesting case is where the functions are polymorphic. Another approach is to associate a typing function with every function. The typing function determines which type constructions are possible.

Mayr: I will discuss the type notion in two steps. First I want to define an ADT, starting with the view that all models are defined by operations. We can get a grip on the objects by describing the way they are constructed. An ADT is defined by a set of operators and a specification in some formal theory; for example, an algebraic specification by axioms, a specification by a set of rewrite rules, or a logical specification such as Hoare's axioms. If you have a set of operators and a specification you can derive the set of objects. So the objects can be described by sequences of operations that construct them. In a second step we can define the notion of type by giving a representation for an ADT or abstract model. This is done using some symbol set with formal symbol combining rules and a representation function that maps the model to the symbol representation.

Carbonell: Why hasn't the notion of type been applied to abstract information about procedures? For instance, one can think of a sort procedure, as one type of procedure, independent of the types of the arguments and the outputs. You can also classify different functions or procedures in a type hierarchy.

Duetsch: There is no reason not to do that. I pointed out in my definition that there was no reason not to apply the same characterization method to procedures as to data. There are lots of ways you can build a type system for procedures. For example, Scott's hierarchy is a type hierarchy for procedures that deals only with the functionality of the procedures.

Rich: At MIT we are building up a library of procedure specifications, with pre-conditions and post-conditions. They are organized in a hierarchy based on the relationships of the pre- and post-conditions. You can add post-conditions or specialize your arguments. You can view instances of that type as being the actual applications of that procedure.

Zilles: One important issue that we have ignored is how to determine if something is an instance of a given type. Many reasons why type systems exist is because there are properties that can be easily determined according to some rule. In the different areas, these rules are different.

Deutsch: I would be disturbed if you want to rule out the type system of RUSSELL, in which objects in principle are not typed at all. Asking the type of an object is meaningless in RUSSEL, since the notion of type is completely textual.

Rich: Finding the type of an object is called recognition in AI. In compilation, you want to know what type a variable is. For other situations you may not be able to ask that question. Types can be a way of keeping track of what your constraints are, or a way of combining information stated in two different places.

SUMMARY

Rowe: This was a difficult session to summarize because so many ideas were presented, not all of which were related to types. I will emphasize the

ways in which types were used. First of all, types are not "GOD" given. There is not one fixed notion of what a type is or how it is to be used. Rather it seems they are tools to be used in the different areas to solve various kinds of problems.

It seems that types are primarily used for classifying objects or entities. How you classify objects can vary quite dramatically. Some people refer to sets of values. Others refer to predicates on symbols. Still others refer to properties of entities on objects, e.g., including operations.

There are at least as many goals for using types as there are users. Types may be used to check that the use of an operation is valid. This view comes from the PL community. Ted Codd also used this notion of type in verifying that joins make sense in databases.

Another goal is to select a specific operation. A third may be to describe information about objects or entities. To illustrate this diversity I have chosen some typical uses from each of AI, DB and PL. In AI, type is used to control the search space. Another example due to Deutsch is that a type is used to give incremental descriptions of real world objects. In the DB community, types are used for checking the validity of operations. They are also used to describe information about objects or entities. In the PL community there is a strong emphasis on binding operations with the ADT and protecting the representation. Types are also used in selecting the particular instance of a generic operation that is required in a given application.

Sowa: I would say that AI also includes all the PL and DB uses of types that you gave.