SOME REMARKS ON DATA TYPES

J.A. Feldman and G.J. Williams
Computer Science Department
The University of Rochester
Rochester, New York

TR28

April 1978

The preparation of this paper was supported in part by the Alfred P. Sloan Foundation under Grant Number 74-12-5, and in part by the National Science Foundation under Grant Number MSC76-10825.
1. Introduction

In current programming languages, data types and the associated type machinery are used for a variety of purposes. We will discuss how in some cases alternatives to ordinary data types may serve better, providing language facilities which are more expressive, and more extensible.

For simple variables, a data type can be characterized as the set of operations specifying the interpretation of a given bit string in the memory of the computer. This approach is taken by [Demers et al., 1978]. The distinction between real and integer is a common example, directly related to the difference in hardware operations for real and integer variables. In a tagged architecture where reals and integers use the same set of operations, there might be a single type number. At a lower level, a type bits could be provided, to allow the interpretation of a bit string simply as a bit string.

Since the number of types which have a distinct, direct hardware interpretation is small and fixed, it seems appropriate to regard them as primitive. The present data type machinery is perfectly adequate for this purpose, allowing each simple variable a single interpretation. The compiler will ensure that we do not mix different types in an incorrect way. It is central to the present conception of data type that the compiler checks type correctness.

This has led to the unfortunate notion that if the compiler is to check something, then the thing to be checked must be a data type. Data types are now being used to encode all those properties of a variable which the compiler checks. This is one instance of a common situation, in which programming language form is restricted by an implicitly held model of a very crude compiler. We are engaged in exploring the implications for language design of explicitly taking account of the current capabilities in optimization and verification. This paper will discuss how the design of the data types in a language interacts with flow analysis, value propagation, assertion handling and so on. Our group is also working on the design of language facilities for distributed computing [Feldman, 1976] and data base systems [Shopiro, 1978], and on automatic selection of data structures [Low and Rovner, 1976].

Consider the subrange type of PASCAL. A variable

\texttt{var v : \{1..100\};}

has a hardware representation (or underlying type) of integer. However it is guaranteed by the compiler to lie between 1 and 100. This type can then be used to define an array:

\texttt{var a : array \{1..100\} of real;}

and since the compiler wishes to guarantee that any access to this array is within bounds, this array can only be indexed by a variable of type \([1..100]\), such as \(v\). The compiler is thus assured that the index is between 1 and 100.

There are several difficulties with this approach. The first is how to deal with subranges of subranges, or overlapping subranges. If we define

\[
\text{var } v2: \ [1..50];
\]

can we use \(v2\) to index \(a\)? The answer is that a coercion, a transfer function from one type to another, may be defined, so that \(v2\) can be coerced to the wider subrange, permitting \(v2\) to index \(a\), even though it is not strictly of the correct type.

Secondly, why should the compiler rely only on type information to determine that an array access is in bounds? If the compiler can obtain information by analyzing the flow of the program, then it is not necessary to restrict array accesses to a particular subrange type. If \(i\) is any integer variable, then in

\[
\text{if } 1 < i \text{ and } i < 100 \text{ then } a[i] \text{ else error;}
\]

the compiler might be able to deduce that the array access will always be in bounds, so that \(i\) need not be of an appropriate subrange type. [Habermann, 1973] explains the difficulties with subrange types in detail.

More generally, what if the programmer wants to associate some complex property, or more than one property, with a variable? Presumably each property (i.e., predicate) can be used to define a new type, just as the predicate

\[
P(i) = 1 < i \leq 100
\]

defines a subrange type. However, suppose that the programmer defines 3 independent properties, such as

\[
\begin{align*}
\text{small}(x) &= \text{abs}(x) \leq 15 \\
\text{odd}(x) &= n \mod 2 = -1 \\
\text{positive}(x) &= x > 0
\end{align*}
\]

In a language like PASCAL, types can be used to represent properties and'ed together, such as small odd or small positive, but each is a separate type. Thus, unless the appropriate coercion is defined, one cannot use a small odd variable in a procedure call, for example, which requires a small argument. Composite types can be combined using the union construct, which has the effect of or'ing properties together. This means that either of the types making up the union may be used where the union type is required. In effect, this defines a pair of coercions to the union type. For example, the coercions:
small -> union ( small, odd )
odd -> union ( small, odd )

would be defined.

Unfortunately, even with three basic properties over a hundred different plausible types can be produced. For example:

union ( small, odd )
union ( small positive, odd )
union ( small positive, small odd )

where presumably any property may also be negated.

This creates even greater problems with generic operators, which are operators associated with different function procedures for arguments of different types. For example:

small + small
odd + odd
positive + positive

might each invoke a different procedure. This might be adequate if three procedures were all that were needed, but what about:

small + small odd
small + odd

Perhaps the first can use the same procedure as

small + small

by invoking the coercion of small odd to small, but what about the second? The variable of type small might be odd, so that the odd + would be correct, or perhaps the variable of type odd is small, or perhaps they are both positive! A profusion of composite and union types could not possibly help; it would multiply the number of cases to be considered alarmingly.

Another problem that arises using PASCAL-like strong typing is that string constants are given a type which specifies their exact length. In consequence, two string constants of differing lengths have different types! It is impossible to write a routine which can take any string as argument, without resorting to:

union ( string(1), string(2), string(3), ... )
This construct is equivalent to the notion "string of any length". It is an obvious failing of strong typing that such a simple construct should be impossible to express. We will discuss how to handle it below.

The problem rests on two fundamental assumptions of the type mechanism: that a variable has just one type; and that types match only if they are identical or can be coerced to be so. But neither assumption is necessary to do the job that types are here being required to do. If a programmer wants to declare that a variable always satisfies some predicate P, or that the actual argument to a procedure must satisfy another, then it should be possible to say just that.

2. Properties of Simple Variables

For this purpose, the notion of a property is introduced. A property is basically a restricted form of an assertion [Feldman, 1976], which applies only to a single variable, and applies to that variable throughout its range. Note that the compiler will check that the property is true, either by proving it or by introducing code to verify it at run time. Properties differ from types in that a variable may have more than one property, and more importantly, the actual argument to a procedure must match the formal argument only by satisfying the required properties, not by strict type equality. For a dumb compiler, this might mean that the properties of the formal would have to be the same as (a subset of) the properties of the actual, but a smart compiler might attempt to match the properties of the actual to the properties of the formal using inference rules of a kind to be discussed below.

Properties may be defined in various ways. The most obvious is to declare the property as a predicate. For example:

\[
\text{property (integer) small (x) = abs(x) \leq 15 ;}
\]

Note that it is necessary to declare that small applies to integers.

The following example shows how properties can be defined, and used either in variable declarations, or to describe formal arguments to procedures. The program will be checked for consistency in the use of properties, and often runtime checks may be avoided.
```
begin

property (integer)
small(x) = abs(x) ≤ 15,
odd (x) = x mod 2 = 1,
positive (x) = x > 0;

var v : small odd integer;

function plus (a1, a2 : small integer) : small integer;
begin
  plus := a1 + a2;
end;

v := plus(v, 1);

At the call on plus, the compiler checks that the actual arguments used are compatible with the formal arguments used in the definition. In this case, v is compatible with a1 by inspection, since both are integers and v's properties include small, the property required for a1. The compiler can check that 1 is small by executing the definition of small given in the property declaration. No code has to be inserted to check compatibility. At the assignment to v, the compiler knows that the result of plus will be small, because this is specified in the procedure declaration. However, code must be inserted to check that the result is odd. The compiler will guarantee that the result of plus is small when the procedure itself is compiled. Since it is not possible to deduce that the sum of two small numbers is small (since it is not always true) this will be checked just before the procedure exit. Note that, by examining the code for plus, the compiler could deduce that plus(v,1) is even, and therefore that the assignment to v will cause a property violation.

Now suppose we have a property tiny:

property (integer) tiny (x) = abs(x) ≤ 3;

Then it would be useful if the compiler knew that a tiny integer was also small. Certainly it is possible that some compilers could work this out from knowledge of arithmetic, but there should be a facility for telling the compiler in case it cannot. This is achieved using an assume statement:

assume tiny => small;

The syntax is just:
assume <proposition> ;

where the proposition involves only properties. The compiler may attempt to prove the assumption correct, and will report a counter example or contradiction, but it will do nothing if compile time analysis fails. This distinguishes it from the assert statement, which is fundamentally a statement about runtime, and hence must be checked at runtime if it is not proved at compile time. The assume statement derives its major importance from its use with compound data structures, discussed in Section 3.

Properties can also be declared without giving them an explicit meaning using a predicate. Suppose that you were writing a program that involved various kinds of fruit, and you wanted to declare a property for each kind of fruit so that the compiler would check that you had not mixed them up. You could define a property just like an enumerated type:

\[
\text{property (integer) fruit = ( orange, lemon, apple, cherry) ;}
\]

This declares fruit to be a property, as well as orange, lemon and so on. It also implies that the properties orange and lemon are disjoint, and that if something has the property orange it also has the property fruit, and that if something has the property fruit if must also have one of the properties (orange, lemon, apple or cherry) that make up the disjoint union. Note that, unlike the case of small, it is not possible to check that an integer is fruit by examining its value. We call small extensional (or manifest) and fruit intensional (or uninterpreted).

A simple property can also be declared:

\[
\text{property (person) big, strong, ugly ;}
\]

which declares three uninterpreted properties without relating them in any way.

We could then define a property in terms of previously defined properties:

\[
\text{property (person) tough = big and strong and ugly ;}
\]

The right hand side of this definition could be any boolean expression involving properties.

There are two problems that arise with intensional properties that are more difficult to handle than with extensional properties. The first is that the programmer must indicate to the compiler how the property is to be propagated through statements and expressions; the second is that explicit facilities must be provided to allow the programmer to define coercions from one property to another.
Propagation rules for conventional types are well known. The sum of two real values is a real value; the sum of two integer values is an integer value, and so on for the other operators. There may be a division operator which takes two integer arguments and returns a real result, or an exponentiation operator which takes a real and an integer and returns a real, but the point is that this is defined by the system; it does not have to be specified by the programmer.

Similarly, certain coercions may be defined by the system, such as:

\[
\begin{align*}
\text{real} & \rightarrow \text{union (real, integer)} \\
\text{and} & \quad \text{integer} \rightarrow \text{union (real, integer)}
\end{align*}
\]

or:

\[
\text{integer} \rightarrow \text{real in the context integer + real}
\]

In the case of extensional properties, defined using a predicate, the compiler may either derive propagation rules from the predicate, or use the predicate to analyze each instance of the property, or compile time analysis may fail and the compiler will insert a run time check to verify that the appropriate predicate is valid. For example, the compiler may be able to derive

\[
\text{positive + positive gives positive}
\]

a priori, without examining any instance. If it cannot, it may at least be able to deduce that, in each instance of x, y and z:

\[
x > 0 \quad \text{and} \quad y > 0 \quad \text{and} \quad z := x + y
\]

implies \( z > 0 \)

On the other hand, it may have to insert a run time check at every instance to verify that

\[
\text{even + even gives even}
\]

if, for example, it is not able to make deductions using modular arithmetic.

We have seen how in the case of extensional properties we can supply hints to the compiler via an assume statement:

\[
\text{assume tiny => small ;}
\]

This has the effect of defining a coercion from tiny to small. It cannot, however, affect the type correctness of the program, since without it the compiler may be able to deduce the implication anyway, and if not will simply insert redundant run time checks. On the other hand, if you assume something which is false, the compiler could miss errors it would otherwise detect. It is the responsibility of the programmer not to assume anything
which is false. This is in contrast to the assert statement, where it is the responsibility of the compiler to verify or check the validity of the assertion.

Intensional properties are somewhat different, since it is impossible to check them at run time, or to generate propagation rules by examining their definition. Some coercions may be generated automatically by the system, for example in the case where a property is defined in terms of others:

```
property (integer) vehicle = car or truck ;
```

From this definition the compiler would generate the coercions:

```
car => vehicle
truck => vehicle
```

This is exactly analogous to coercion in the case of union types.

Further coercions may be defined by the programmer in exactly the same way as with extensional properties:

```
property (integer) plant, tree ;
assume tree => plant ;
```

A more complex case arises when a coercion from one property to another must be accompanied by some sort of conversion. For example, two properties inch and centimetre might be defined. Then if an inch variable was used in a procedure call which required a centimeter argument, the variable could be coerced by multiplying by 2.54. This can be handled in three different ways.

A transfer routine can be coded, and used explicitly whenever required:

```
begin
property (real) centimeter, inch ;
function InchesToCm ( x : inch real ) : centimeter real ;
InchesToCm := x * 2.54 ;
procedure Funk ( y : centimeter real ) ;
begin . . . end ;
var a : inch real ;
Funk ( InchesToCm (a) ) ;
end ;
```
The second alternative is that the possible coercions could be declared in the argument list, along with the necessary transfer functions:

```
begin
  property (real) centimeter, inch;
  function InchesToCm (x : inch real) : centimeter real;
  InchesToCm := x * 2.54;
  procedure Funk (y : centimeter real or inch real coerced with InchesToCm);
  begin • end;
  var a : inch real;
  Funk (a);
end;
```

The third alternative is that possible coercions could be globally defined:

```
begin
  property (real) centimeter, inch;
  coerce inch to centimeter with InchesToCm;
  function InchesToCm (x : inch real) : centimeter real;
  InchesToCm := x * 2.54;
  procedure Funk (y : centimeter real);
  begin • end;
  var a : inch real;
  Funk (a);
end;
```

This last alternative seems the most natural, is not hard to implement, and seems to add greatly to the ease of writing reliable programs.
For general propagation rules, however, the situation is more complex, since conceivably the result of every operator on every possible pair of properties might have to be specified. In practice two things will help. First, most combinations of properties and operators will be illegal, and could be left unspecified by allowing a default rule which signalled error. Second, the system might provide commonly occurring sets of propagation rules which could provide a shorthand for the rules spelled out in full.

Consider, for example, properties to be applied to integers recording counts of various fruits, as in our declaration of "fruit" above. The propagation rules are quite obvious: apples plus apples is apples; apples minus apples is apples; apples times apples is an error, but apples times (not fruit) i.e. apples times integer, is apples. If it is allowed, apples plus oranges has the property fruit.

These propagation rules can clearly be applied to any counting situation, and so could be encapsulated in a set of rules provided by the system. To illustrate a plausible syntax, we present another set which might be provided by the system. As before, it is assumed that any combination not mentioned is illegal.

\[
\text{propagation rules} \\
\{ \\
miles + \miles \rightarrow \miles \\
miles - \miles \rightarrow \miles \\
miles / \miles \rightarrow \text{none} \\
miles * \miles \rightarrow \text{sqmiles} \\
\text{sqmiles} + \text{sqmiles} \rightarrow \text{sqmiles} \\
\text{sqmiles} - \text{sqmiles} \rightarrow \text{sqmiles} \\
\text{sqmiles} / \text{sqmiles} \rightarrow \text{none} \\
\text{sqmiles} / \miles \rightarrow \miles \\
\} \\
\]

which could just as easily be expressed by:

\[
\text{propagation rules} \\
\{ \\
\text{dimension (1 : miles, 2 : sqmiles)} \\
\} \\
\]

The following example shows how intensional properties may be defined, how propagation rules work, and how generic procedures may be defined using properties.
begin

property (real) precise, approximate;

propagation rules
{
    precise any precise -> precise;
    precise any approximate -> approximate;
    approximate any precise -> approximate;
    approximate any approximate -> approximate;
}

generic procedure Print (x);
    when precise real x do
        "print out x to 8 places"
    when approximate real x do
        "print out x to 3 places";

var temp: precise real;
var velocity, effectivetemp: approximate real;

input (temp); input (velocity);

effectivetemp = 50 - (50 - temp)*(1 + .01*velocity);

print (temp);
print (velocity);
print (effectivetemp);

end;

Here the propagation rules are applied in the computation of "effectivetemp", to verify that it is "approximate". Then the properties of "temp", "velocity" and "effectivetemp" are used to decide which particular "print" function to use.

For some intensional properties, it is not possible to give explicitly the set of propagation rules which would allow the compiler to verify that the property is being used consistently. For example, the property might be associated with a complex data structure, so that no single operation would be sufficient to, say, update the data structure and change its properties. In order to handle such properties, it is necessary to introduce the idea of promotion. Suppose a data structure is in an intermediate state where none of a set of properties applies to it. Then at some point the data structure enters a consistent state and some property becomes true. We say that the structure has been promoted (i.e., to the state where the property is true). The following example shows how this works. We consider the properties precise and approximate, as above, but in a situation where propagation rules do not help. Promotion is expressed using the assume statement. This use is consistent with the previous use of the assume statement to provide the compiler with compile time information not given in other ways.
The assume statement is necessary, since otherwise the assignment of `val` to `SquareRoot` would be a property error. The compiler will check that the arguments to `SquareRoot` have the correct properties, and that all assignments are legal, in the sense that the right hand side of an assignment has at least those properties required for the left hand side.

Strong typing requires that all compile time information about a variable be encoded into its data type, making it very difficult to express different sets of constraints for different variables. Defining and using general procedures is hard, because the minimum requirements for an argument will be encoded into its type, which must match the type (i.e. encoded properties) of the actual. Thus a general procedure cannot be used with actual arguments that are more tightly constrained than the formal arguments.

Properties provide a convenient way to express constraints on variables, since a variable may be declared with a number of properties. It is easy to define and use general procedures, since each formal argument may be declared with just that set of properties necessary for the procedure to work correctly, and the procedure may be called with any arguments that satisfy those requirements. Assume can be used to give the compiler extra information about relationships among properties, such as
possible coercions from one property to another.

The property mechanism provides two different kinds of information for program verification. Extensional properties aid conventional flow analysis by giving the verifier statements about a variable which are always true throughout the lifetime of the variable. This is helpful even though they might still have to be verified, since they function as hints as to what properties of a variable are significant in the operation of the program. Intensional properties, on the other hand, are checked for consistency rather than validity. They are used by the programmer in the same way that casting out nines will help to verify an addition; an inconsistent result indicates an error, but an error will not necessarily produce an inconsistency. Note that we are not describing a scheme for verifying that a program meets a set of abstract specifications, possibly expressed in some specification language. We are concerned with allowing the programmer to express the constraints he has in mind when the program is written, so that the compiler may verify that the constraints hold, and use them to produce efficient code.

3. Structures

We would now like to describe how to extend properties to cover compound data structures, as well as simple variables. If properties are useful for simple variables, then surely they must also be useful for data structures. Just as the notion of "number of oranges" may be expressed as an integer variable and a property "orange", the notion "address" may be expressed as a string of characters and a property "address". This would prevent any other kind of string from being used where an address was required. More importantly, if the structures themselves were defined using properties, then procedures which would work correctly on different structures could be defined by expressing the constraints on the arguments with structure properties. A clear example of this is a procedure which works on a particular structure, but does not depend on the type of the components of the structure. For example, a procedure to insert an element into a list will work correctly on a list with elements of any type.

Attacking the same problem, Mitchell and Wegbreit [Mitchell and Wegbreit, 1977] propose the idea of a scheme, which is a data structure and the procedures needed to access it, defined with a type parameter. For example, a scheme for list could be defined, which would be parameterized by the type of the elements of the list. This has, however, merely pushed the problem of strong typing back one level. What if, as mentioned earlier, you want to define a routine which will take a string of any length as argument? You can certainly define a routine:

```plaintext
procedure PRINT (t : type, str : t) ;
```
with a type parameter, which can be filled in when the procedure is called, but you have lost the information that "str" must be a string. Mitchell and Wegbreit point out that this problem can be solved by adding a specification to "t":

```pascal
procedure PRINT (t : type needs ..., str : t);
```

and point out that the needs clause can specify any boolean expression which can be evaluated at compile time. The proposal in [Demers, 1978] and the examples given by Mitchell and Wegbreit essentially restrict this boolean to testing the presence of operators or procedures necessary for the procedure to work correctly. The property mechanism proposed here provides a way to generalize this idea, by allowing the needs clause to specify any set of properties.

It remains to show that it is possible to define data structures using properties in a way that will provide the same advantages as were possible with simple variables.

The lowest level of a data structure, roughly analogous to the hardware representation or underlying type of a simple variable, is the dimension. A simple variable (the simplest data structure!) has dimension 0, a vector dimension 1, and a matrix dimension 2.

We will use row to mean a 1-dimensional structure, row of row a 2-dimensional structure, and so on. The use of "row" to describe a structure is not meant to imply anything about it except its dimension. An array, a record, and a list are all "rows". We will use simple to mean any simple type. It includes primitive types like real, integer, or boolean, as well as PASCAL-like enumeration types, such as:

```pascal
type color = ( red, amber, green ),
           direction = ( north, east, west, south );
```

Two more concepts will suffice to define the bare skeleton of a data structure. Const means that the structure (or element) is fixed at compile time. Note that const row means that the size and storage of the structure is fixed; it does not mean that the values of the elements are constant. A constant is const simple, a FORTRAN array is const row of simple, and a table of constants is const row of const simple. Ref means pointer, and is used to distinguish between the two possible semantics for

```pascal
B := A;
B[I] := new value;
```

If the variables A and B are declared as row then B will be a separate copy of A, and A[I] will not be changed; if A and B are declared ref row, A and B will point to the same object and A[I] will be changed.
At the next level of description a data structure will be defined so that a particular representation is uniquely determined. A particular compiler may support a large number of different representations, or it may only, like FORTRAN, support const row. In the same way, the number of properties used at this level may be large or small. We will present a reasonable set, enough to define most 1-dimensional data structures. It is not intended that this scheme cover such structures as graphs or trees, which are not factorable into 1-dimensional structures. Multi-dimensional structures are covered only by treating each dimension separately.

The properties we will use to exemplify this level of description are:

i) fixed / varying (with respect to length)

Fixed length means either fixed at compile time (such as PASCAL arrays), or fixed at object creation time (such as ALGOL '60 "dynamic" arrays). There is no compelling reason for not distinguishing between these two cases. A compiler could perfectly well provide two separate properties instead of just "fixed". The set of properties which we present is adequate for our purposes, but we make no claim to its optimality.

ii) uniform / mixed (with respect to type)

Uniform means that an access to the structure will produce an element of the same type, independent of which particular element is accessed. Here "same type" means identical with respect to this level of description, or for simple components, identical with respect to primitive type. Thus two vectors have the "same type" if the type of their components is the same. This does not depend on their length, which is not defined at this level.

iii) index / name / seq / assoc / none (accessing)

A structure is accessed by index only if it is indexed over a small set of fixed size. This is exactly the same as a PASCAL array. Accessed by name means accessed using a fixed set of names, known at compile time, with all components always present. This corresponds to a PASCAL record. Assoc means accessed by specifying a name or value which need not be declared at compile time, and which need not be present in the structure at any particular time. This is just an associative data structure, such as a SNOBOL table. Seq means accessed using a "next" operator. A component can only be accessed by chaining down from the head of the structure. This corresponds to a linked list. None means that a component may be inserted or removed but not changed in place.

It is convenient to assume that certain functions are automatically defined on structures with given properties. For example, the operations associated with each of the properties given above might be:
index indexed read/write
name indexed read/write
assoc associative read/write, insert/remove
seq read/write, insert/remove, next
none insert/remove

Note that these properties are not independent. For example, an indexed structure always has fixed length. On the other hand, some properties which distinguish data structures are not on this list. One example is:

"S contains no more than one occurrence of any value."

which serves to distinguish sets and bags (multisets).

Now we can characterize several common data structures:

list: varying, mixed, seq
vector: fixed, uniform, index
record: fixed, mixed, name
table: varying, mixed, assoc

Here table refers to a SNOBOL table, which is associative. A particular language could, of course, require that its lists be homogeneous (like SAIL), or allow its records to be indexed (like ALGOL '68).

Some of the above properties are related to the set of operations which are possible, or efficient, using the data structure. For example, variable length implies an insert and a remove operation, and indexing is itself an operation. But we are by no means defining data structures in terms of the set of operations on them. For example, the type of the components and the length of the structure are quite unrelated to any operation to be performed on the structure. Furthermore, similar but distinct operations, such as insert and remove for lists and push and pop for stacks are not distinguished in any way. Thus the set of properties does not define an abstract data type.

On the other hand, the set of properties do not alone force a particular implementation. The compiler is free to choose between a linked list and a boolean array to implement a set, for example. The properties of the data structure are certainly powerful hints, and a dumb compiler might use the obvious implementation for each data structure, but the compiler could examine other assertions in the program, or deduce how the data structure is used by analyzing flow of control within the program. This technique is explained more fully in [Low and Rovner, 1976].

The third level of definition, one which corresponds to the notion of type, is the filling in of the values of the attributes which have been defined as specified at compile time. Thus, for example, the length of a vector and the type of its components will be specified, as will the names and types of the fields in a
Thus there are three distinct levels of definition. The first is the bare skeleton, defined using `simple`, `row`, `const`, and `ref`. The second level adds properties to define a particular data structure, and the third level includes the specification of all details which are fixed at compile time. It is also possible to attach to any type or variable a set of assertions which serve to define it further, such as the assertion that a set contains no duplicates, or properties like `small`, `odd` and `positive`.

We will now give several examples of the definition of a type at each of the three levels. A syntax similar to that of PASCAL will be used. In PASCAL:

```pascal
type complex = record re, im : real end ;

var z : complex ;
```

defines a type constant "complex" and a variable "z". Alternatively, z could also be defined by:

```pascal
var z : record re, im : real end ;
```

We will use the keyword `structure` to mean a definition at level one or two, and `domain` to refer to a definition at level three. `Type` will mean a domain with a defined set of operators and functions. This is because type is defined to be the set of operations which specify the interpretation of values over some domain [Demers et al., 1978]. For built-in domains, like `integer`, the set of operations is defined by the language. We will show later how to handle user-defined domains. In some cases, the type will be produced by giving the domain a default set of operations. For example, a particular record type might be just the underlying domain with the operators for assignment to a record, assignment to a field, and field selection.

A) A Hoare record

The three levels of definition are:

i) `structure Hrecord = const row of simple ;`

ii) `structure Hrecord = const row (fixed, mixed, name) of simple ;`

iii) `domain date = Hrecord [`

```
    month : integer ;
    day : integer ;
    year : integer ;
]`
B) A two-dimensional FORTRAN array

i) \textit{structure} Farray = 
   \begin{align*}
   &\text{const row} \\
   &\text{of const row} \\
   &\text{of simple} \\
   \end{align*}

ii) \textit{structure} Farray = 
   \begin{align*}
   &\text{const row (fixed, uniform, index)} \\
   &\text{of const row (fixed, uniform, index)} \\
   &\text{of simple} \\
   \end{align*}

iii) \textit{domain} picture = Farray (256,256) of integer \\

C) A list

i) \textit{structure} list = row of simple \\

ii) \textit{structure} list = row (varying, mixed, seq) \\

of simple \\

iii) \textit{domain} reclist = list of (atom or ref(list)) \\

D) An Aset

An Aset is a set of name: value pairs analogous to the LISP A-list. Further details can be found in [Feldman, 1978].

i) \textit{structure} Aset = row of simple \\

ii) \textit{structure} Aset = row (varying, mixed, assoc) \\

of simple \\

iii) In a strongly typed language the possible names and their corresponding types would be declared at this level.

E) A table of constants

i) \textit{structure} table = const row of const simple \\

ii) \textit{structure} table = 
   \begin{align*}
   &\text{const row (fixed, uniform, index)} \\
   &\text{of const simple} \\
   \end{align*}

iii) \textit{domain} inttable = table(5) of integer \\

This type could be used in a constant declaration:

\texttt{const powers : inttable = (2,4,8,16,32)} ;

or alternatively:

\texttt{const powers : table(5) of integer = (2,4,8,16,32)} ;
We will now consider how to deal with data structures that cannot be fully defined using our simple one dimensional scheme. A digraph is a representative two dimensional data structure, but it cannot easily be expressed as a row of row structure. We will consider how to define a binary tree, which is a restriction of a digraph.

A binary tree can be represented using a set of nodes of the form:

\[
\text{domain node} = \text{const row}
\]

\[
\begin{align*}
\text{datum} & : \text{integer} ; \\
\text{lson} & : \text{ref(node)} ; \\
\text{rson} & : \text{ref(node)} ;
\end{align*}
\]

which is just a Hoare record. A tree will therefore be represented using the domain nodeset:

i) \text{structure recordset} = \text{row of const row of simple} ;

ii) \text{structure recordset} = \text{row (varying, uniform, none)}

\text{of const row (fixed, mixed, name)}

\text{of simple} ;

iii) \text{domain nodeset} = \text{recordset of node} ;

Four further assertions are required before this definition makes sense as a tree: each node occurs only once in the tree (i.e., it is a set, not a multiset, of nodes); each ref(node) is null or points to another node within the tree, which makes it a graph; there is a distinguished node, the root; and each node has a unique predecessor, except the root.

If the assertion language were powerful enough, these conditions could be expressed as an extensional property. Then the verifier could check that any procedure which created or manipulated trees preserved the tree property, and that the rest of the program was consistent for that property. It is clear from the Alphard project [Shaw et al., 1977] that we do not know how to include in a compiler a verifier capable of handling extensional properties like "tree". The conditions for a tree therefore have to be expressed using intensional properties. In the following example we use an intensional property to encode the four conditions for a tree, and show how this is used with parameterized types, and the definition of a type as a domain and a set of operations [Demers et al., 1978].

\[
\text{domain node of t} = \text{const row}
\]

\[
\begin{align*}
\text{datum} & : \text{t} ; \\
\text{lson, rson} & : \text{ref (node of t)} ;
\end{align*}
\]
domain nodeset of t = set of node of t ;

property (nodeset) tree ;

type bintree of t =
    tree nodeset of t with 
    [ insert : proc (bintree of t, t) ;
      remove : proc (bintree of t, t) ;
      member : proc (bintree of t, t) : boolean ;
    ] ;

Thus a particular type of bintree could be defined by specifying a particular type for the formal parameter. Procedures of the appropriate type, with the names "insert", "remove" and "member" would also have to be defined or a type error would result.

Since a "bintree" is a "nodeset" with the property "tree", a "nodeset" variable can be promoted to a "bintree" using an assume statement. It can then be assigned to a "bintree" variable:

```
var plum : bintree of integer,
    S : nodeset of integer ;

assume S tree ;
plum := S ;
```

Without the assume statement, the assignment would be illegal.

Since "tree" is defined as an intensional property, the compiler will not check that "tree" variables actually have the properties necessary for a tree. The programmer would have to satisfy himself that any procedure which created or manipulated trees was correct, in that the tree property was assumed only when it was true. Within these procedures, one could include assertions (cf. [Feldman, 1976] and Euclid [Lampson et al., 1977]) that helped establish the desired property. These assertions, which are just the uncheckable intensional property translated into program-specific checks by the programmer, will be checked by the compiler. Then the program would be checked for consistency by the property mechanism. In effect, the programmer has defined an extensional property for himself, which he will verify, and is communicating this to the compiler using an intensional property. This is still much better than nothing at all, since the programmer only has to verify those sections of code where trees are being manipulated; the rest of the program will be checked by the compiler. Moreover, the declaration of an intensional property provides an indication that the program is relying on some unspecified conditions for its validity.
4. Conclusion

This paper is a preliminary report on one aspect of the Rochester Advanced Compiler project. The ideas presented here are being incorporated into a base language, Zeno, and an analysis scheme for its compiler. Although there are a large number of issues yet to be resolved, the current set of ideas seems sufficiently coherent for us to solicit comment. Among the open questions are:

i) The semantics of the assertion language, which covers assertions, extensional properties, and assume statements.

ii) How to encapsulate a data structure so that all accesses or uses of the structure can be guaranteed to preserve the necessary properties.

iii) Procedures are normally typed by the type of their arguments, and the type of the result. We would like to describe procedures using properties.

iv) The details of how type parameterization, sketched in the last example, should work.

v) How to extend the property mechanism to systems of parallel processes, so that both processes and their interactions can be characterized by properties.
References

[Demers et al., 1978]

[Feldman, 1976]

[Feldman, 1978]
Jerome Feldman, "High level language constructs for distributed computing," to appear in Communications of the ACM.

[Habermann, 1973]

[Lampson, 1977]

[Low and Rovner, 1976]

[Mitchell and Wegbreit, 1977]

[Shaw et al., 1977]

[Shopiro, 1978]
Jon Shopiro, "A programming language for relational data bases," Computer Science Department, University of Rochester, TR31.