David B. Lomet

A Data Definition Facility Based on A Value-Oriented Storage Model

A data definition facility is presented that provides a consistent description of both primitive and user data. It is based on a value-oriented storage model which carefully distinguishes between values and objects. It is values that are typed in this model, and operations of the type work explicitly on the values. Objects are accessible only via reference values. Objects are described via descriptors called templates, which ultimately yield reference type values. Operations, both primitive and user-defined, are part of a "machine interface," and all executable language constructs can ultimately be defined as explicit operations of the interface. Importantly, these operations must respect the typing constraints imposed by both the primitive types and the user extensions. The interactions of definition facility, storage model, and execution model are illustrated via a series of examples in which commonly used data constructs are defined.

1. Introduction

Several recent languages with data definitional facilities [1-5] have been described in terms of an object-oriented storage model. In this model, all data are treated as storage objects, and these objects are directly accessible to operations. Values do not exist separately from objects. This object-oriented view is a recent development. Conventional programming languages, *e.g.*, ALGOL 60, PL/I, COBOL, FORTRAN, while frequently lacking explicitly stated models, nonetheless share an underlying intuitive model that is quite different from the object-oriented view. In this "value-oriented" view, values and objects are truly distinct and operations manipulate only values. Where reference (pointer) values exist in the language, objects become accessible via the manipulation of these values. Even without explicit reference values, parameter passing is frequently described as "call by reference" to indicate that a reference value is being passed to a subroutine so that the referenced object can be manipulated.

We intend to make explicit this intuitive, value-oriented storage model, but, in addition, to describe a data definition facility which permits users to define their own data in a way consistent with this model. This is the reverse of what has happened, in our opinion, with the object-oriented languages. There, a model understood in terms of the data definition facility was extended into the realm of the primitive data of the base language. The result, we believe, is an unfamiliar storage model and an unnatural programming language.

Our goal here is not to describe some specific programming language. Rather, it is to provide a framework in which most procedural languages can be described. This framework integrates three facets of programming languages that have resisted previous attempts at unification. They are:

Storage model

The value-oriented model, which carefully distinguishes values from objects, builds on our previous work [6, 7]. This model differs from our prior work in its treatment of values as being typed. It is, we believe, an explicit characterization of the intuitive storage model behind most procedural languages.

Execution model

It is important that user-defined operations be structurally consistent with primitive operations. For this reason, we put forward an execution model in which all computation is the result of the execution of explicit operations at a "machine interface." Operations have external constraints that include the specification of their argument and return types. User-defined extensions, *e.g.*, proce-

dures, have the same forms of constraints and can be used interchangeably with primitives.

Data definition facility
This facility, which is the main subject of the paper, permits users to define data that are describable in terms of the storage model used for the primitive data of the base language. Moreover, these definitions are in terms of operations consistent with our execution model, the primitives of which define the base language. This uniformity of primitives and user extensions is assurance that our underlying models, both storage and execution, have indeed captured the essence of the primitive constructs. In particular, we believe that the data definition facility captures the essence of the term "data type."

The remaining sections of the paper pursue, in considerable detail, the way that these three facets interact. The next section provides an overview of the three areas, while the remaining sections explore the data definition facility and its integration with storage and execution models. Examples are extensively used so as to provide convincing evidence as to the success of the approach.

2. Framework

• *Storage model*
In the object-oriented view, an object typically possesses three attributes: a location, a value, and a type. All data are modeled as instances of objects. Values are not directly accessible to operations. Rather, a value is solely state information associated with an object. One never "sees" a naked value. Because objects have locations, all objects can be defined with update operations and hence are, at least conceptually, amenable to being changed. Constants (values) in this view are immutable objects, *i.e.*, objects without update operations.

The value-oriented view turns most of this around. In this view, objects and values are truly distinct. Objects, which may contain values, are described by means of descriptors that we call templates. Objects can only be manipulated via their reference values. It is values that are typed and that are directly accessible to operations. Types describe values in that they specify the representation of values of the type and the operations permitted on these values. A type is not merely a tag associated with an object (cell) that restricts the values that can be stored in the object to some subset of a universal domain of values. Rather, the type associated with a cell now identifies the unique type of the values that can be stored. This distinction between types as sets of values *versus* typed values is most clearly illustrated in the case of unions. Type as a set specification associated with a cell suggests that there are not values of a union type but rather that a cell can contain values from more than one type. Our new view requires that there be union type values and further that a value drawn from one of the alternative types of the union be explicitly converted to a union type prior to this value being stored in the cell.

Example **int** ∪ **char** does not define a set union of integer and character types; rather it specifies a completely new type that happens to be able to represent values drawn from both alternative types, but, in fact, does not support the operations of either type.

An object, in our view, has a fixed set of components, *i.e.*, parts to which a reference value can refer. Unlike types, templates do not take part in unions. Thus, it is not possible for an object to have a varying number of components, the number and form of which change depending on the current value(s) contained by the object. Objects with varying component structure have been called "flexible." In ALGOL 68, [1:0 **flex**] **int** is such a flexible object. ALGOL 68 precludes references to components of such flexible objects, but only by means of an *ad hoc* rule. The object/value distinction, in which objects have a fixed structure and templates do not take part in unions, supplies a unifying rationale for such a rule. At the same time, a great deal of flexibility can be provided by the type system, *e.g.*, in the form of unions of aggregate values, because this unioning applies to types but not templates.

Note that to model most existing languages, one must provide aggregate objects whose components are truly part of the aggregate, *i.e.*, whose existence depends on the existence of the containing aggregate. These aggregates do not refer to their components. Rather, the components are included as a part of the object. The updating of a component does not change the identity of the component so that it now becomes a different referenced object. An update merely changes the state of the component. While physical contiguity of representation is not required to sustain this notion, the clear distinction between what exactly comprises the object *versus* what is merely referenced via the object must be maintained.

A duality between objects and values can be usefully exploited, and we shall do this. This duality permits us to use a template to describe both an object and an analogous value. Not only is an economy of description achieved by this, but the process of generating aggregates can be much simplified. This will introduce some extra complexity into the template but appears to avoid the even greater complexity of providing dual forms of operations for constructing both objects and values. The template definition, which with objects distinguishes between

Table 1 Properties of values and objects.

	Property	*Discussion*
Values	atomic	Referencing components is not possible.
	immutable	Values may be replaced by other values, but it is meaningless to talk of changing values. If 1 is added to 3, the effect is not to change 3 to 4 but to replace 3 by 4.
	storable	Values can become the contents of (parts of) storage objects.
	accessible	Operators take values as arguments and may produce values as results. Further, any result returned is always a value.
	typed	Each value has a unique type that determines the operations which can access its representation.
Objects	constructable	Whereas operators may return only values, a side effect of certain operators is the creation of storage objects that persist over time.
	referable	Reference values are used to specify the storage objects or their components that are to be manipulated.
	changeable	Storage objects contain the state of the computation. If an object is changed, subsequent operations involving references to it reflect the change.
	deletable	Some languages permit storage objects to be deleted. Subsequent references to the deleted object are erroneous.

what is part of an object and what may be referenced by the object, serves here to separate that which is in the representation of the value, and hence cannot be altered, from that which may merely be referenced, and hence continues to exist as an independent object that can be updated.

The properties of objects and values are summarized in Table 1, which is a modification of a similar table in [7]. The most notable change is the inclusion of the "typed" property for values.

- *Machine interface*

One long range goal of this work is to provide a complete machine interface in the same sense that, *e.g.*, the IBM System/370 interface is complete. Current high level languages do not do this. Thus, *e.g.*, ALGOL or PL/I requires a linkage editor and a system command language. It is not possible with these languages to specify that programs are to be compiled, linked, loaded, and executed. A complete machine interface provides the only way of dealing with a computer and hence must make all of these currently excluded functions possible.

Our intent is for this interface to be low level in all its aspects except for the procedure and data definition extension mechanisms. The low level nature of the primitive operations avoids the primary problem with high level machines, *i.e.*, the large granularity of the operations, which makes it difficult to efficiently support other high level operations that differ only modestly from those provided as primitives. The extension mechanisms provide a means of specifying efficiently the forms of high level data and high level operations that each user may require.

Like a conventional machine, all computation is performed by some explicit operation. Thus, every type conversion, every address computation, every update, every control structure is realized by an explicit operation, whose arguments and results must be values supported by the interface. Note that most object-oriented approaches are not compatible with this requirement.

An operation is characterized, in its interactions with surrounding operations, by its argument types and its return type, if any. These type specifications are precise. That is, the exact type of the argument or return type is prescribed. Thus, within an operation there need be no type checking of parameter/argument matching. Further, only operations specified with a parameter's type definition can be used to manipulate the argument. In this machine, all type checking is done during the program construction process, via program construction operations. This construction process requires that, during the combining of operation with arguments in the resulting program, each argument exactly match its corresponding parameter. Where such a requirement is burdensome, a program translator, which takes character strings and creates programs, must supply the missing operations that are implicit in the character string form of the program. The interface itself does not provide this service. Likewise, the interface does not provide polymorphic operations, *i.e.*, those whose argument or result types depend on other argument types. Thus, polymorphism as well is treated as an issue for the translator. When examined closely, most languages or language proposals treat polymorphism this way [1, 2, 5, 8, 9].

Why belabor this point concerning explicit operations and exact type matching? Because machine interfaces require the explicitness and because current languages, with the exception of LISP [10], which is a typeless lan-

guage, are not amenable to such a view. For example, the array element denoted by "A[i]" in ALGOL 60 [11] denotes the *i*th element of array A, where *i* is specified at the time the element is needed. Explicit operations, in our view, would treat this as an expression containing operations for computing the *i*th element given *i* and a reference to array A. In ALGOL 60, neither the reference values nor the selection operations for computing the location of the *i*th element are included in the language. ALGOL 60 is not unique in this. We are not aware of any typed languages which lend themselves to an interpretation as a sugared form for such a machine interface.

Why might such a machine interface be of interest? There are at least two reasons:

1. Such an interface can be construed as the lowest level "machine-independent" target for a compiler. Compiling to this level provides two advantages. One is that efficiency can be achieved in the resultant code because all the operations can be low level ones with small granularity. The operations are all exposed, and thus common forms of optimization can be profitably applied to the program in this form. Two, the interface remains type secure, and hence the type system cannot be compromised by errors in compilers, which tend to be large programs that are rarely error free.
2. It should be possible to implement secure systems on top of this interface in a manner that is cost competitive with existing, nonsecure systems. Capability machines/systems, which traditionally have been the base on which secure systems are built, have cost/performance problems stemming basically from the requirement to protect pointers and the complexity and cost of the domain switching operations. In our interface, the type checking inherent in program construction guards pointers, indeed all data, from the danger of being misinterpreted. Further, the interaction of type checking with procedures forms a natural basis for restricting addressability. An oft repeated truism is that if only everyone had access to the machine solely through a high level language, then most protection problems could be solved. Our type checked machine interface permits us to solve protection problems in a similar way.

• *Data definition facility*

Our goal with the data definition facility is to provide a uniform way of regarding both primitive and user-defined data. The facility should be sufficiently flexible so that the forms of data in most existing languages can be described. Given our value-oriented view, both values and objects need to be capable of definition. The requirements for our data definition facility are

1. A new type of value must be definable in terms of an existing type. We believe this should be accomplished without introducing a new intermediate object, which would result in an extra level of indirection for every level of definition. So far as we have been able to determine, both CLU [2, 12] and Alphard [5] require such levels of indirection, though it is sometimes possible for an optimization to remove them. Thus, values of type "int" should merely re-interpret the values of the primitive type "bit_32," not add a layer of indirection as well. How this is accomplished is described in the next section.
2. New types of values should be able to play the same role as the old types. Thus, because cells can be defined that contain "bit_32" type values, it must be possible to also define cells that can contain values of type "int." Further, to insure that we have omitted no capability of the primitives in providing our extension mechanism, we require that it be possible to redefine the primitive types using the definition facility.
3. Templates that describe new forms of objects must similarly be part of the definition facility. Requirements similar to those for the type definition mechanism apply also to the template definition mechanism. Thus, user-defined templates must be usable in the same contexts as primitive templates. Further, the primitive templates must be replaceable with user-defined versions that are indistinguishable in effects from the primitives.
4. A means must be provided for conveniently supporting aggregate values, *e.g.*, array values in which the array components have values but not locations. With our approach, aggregate value types can be generated from the primitive templates. For consistency, it is required that new templates be capable of maintaining this object/value duality. This means that template definitions must be flexible enough to contain the type specifications not only of reference types but also of the aggregate value types as well.
5. Type definitions must be specified in terms acceptable to our machine interface. Thus, all operations of a type must be explicitly given. These operations must be described so as to form compatible extensions to the primitive operations. Hence, they must be characterized by their argument types and their return type. These user-defined operations will then be susceptible to the same program construction process as was used with the primitive operations.

The remainder of the paper elaborates the data definition facility and its consistency with both the value-oriented storage model and the machine interface execution model. Many of the common forms of data in the widely used procedural languages are treated in the examples.

Both primitive data and the user-defined extensions are presented so as to emphasize the commonality of treatment. Our success can be assessed by how satisfactorily these common data forms are handled.

3. Types and values

A type not only specifies a set of values (of that type) but also the operations permitted on those values. The values of each type form disjoint sets, in contrast to the purely set view in which a value can be a member of more than one type. A type definition requires the specification of

1. The representation for the values of the type being defined. This representation is specified in terms of an existing type, called the representation type. Each value of the type being defined has a representing value that is a value of the representation type.

   Note: Both the instances of the defined type and the instances of the representation type are values, not objects. There need be no explicit storage associated with instances of a type.
2. A set of ⟨name:operation⟩ pairs. The operations are associated with the type being defined and are the only operations that are permitted to directly manipulate values of the type. These operations are able to do this by gaining access to and manipulating the corresponding values of the representation type. The "name" component identifies the operation that a user of the type desires.

We have not discussed three important features of types.

Operation specification
How do we specify which operation we intend to use from among the several given with the type? We are not particularly concerned with syntax here. What we wish to determine is what operation is used to acquire an operation from a type. What we suggest is that a type also be considered as a function. When used as a function, it takes operation names as arguments and returns the corresponding operations as results. (This view is not essential, but is convenient.)

Example Let "XYZ" be the name of a type and "qrs" be the name of an operation of that type. Then, evaluating "XYZ (qrs)" will yield the qrs operation of that type.

The operations are derived from the type and are not regarded as components of an instance of the type. This is similar to the view taken by CLU where, when a stack is defined, "stack$push" denotes the push operation. Thus, the operations are available as soon as the type is specified. This implies, of course, that the operations are not specific to any instance of the type as they are in SIMULA [13]. Any value to be operated upon must therefore be passed to the operation explicitly as an argument.

Accessing the representation
How do the operations in the type definition gain access to the representing value (instance) of the defined type? This is accomplished by means of a function called **rep** that is automatically provided by the type definition mechanism. The function **rep** takes an argument of the defined type and returns a result of the representation type. This result is the representation of the argument value of the **rep** function.

Instantiating type values
How are instances of the defined types generated? Another function provided automatically by the type definition mechanism is **abs**. The **abs** function takes an argument of the representation type and returns a value of the defined type. This result is the "abstract" value of the defined type that has the argument value as its representation.

Both **rep** and **abs** are made available only to the named operations of the type definition. Their operational effects are available outside the type definition only if the type definition includes operations that supply them. Thus, as much protection or flexibility as is desired can be provided. One more point is that no operations are local to the type definition aside from **rep** and **abs**, nor can a type definer specify them. If such operations are desired, one must augment the representation type so that these operations become part of it.

It should be noted that aside from **abs**, no special construction operations that allocate and initialize storage are provided. This being the case, it is necessary for named operations to exist in the type definition that yield values of the defined type without requiring arguments of the defined type. These operations are not treated specially, however, and there is no restriction as to the number of such operations. Note that this is not the view taken almost everywhere else [2, 5, 9]. This places an additional requirement on declarations. Since no special construction specification is given, declarations must include the means of generating the values that are identified with the symbols of the concrete language. This can be accomplished in much the same ways as in ALGOL 68 [14, 15]. An expression must be present in the declaration. Its result is identified (see the ALGOL 68 identity declaration) with the declared symbol.

When it is intended that a type definition specify a set of references to storage objects, the representing value of each such reference must itself reference some storage object. We describe the primitive types and type generators that initialize this process. They too have type definitions that are of the same form as described above.

Before proceeding to the examples, we present some notation that will be used in the type definitions.

Notation 1 The interface of an operation is described in terms of its argument types, result type, and characteristics as follows:

λ(⟨list of parameters⟩) {**proc** | **funct**} (⟨result type⟩)

Thus λ([⟨identifier⟩:]int,[⟨identifier⟩:]char)**funct**(int) denotes an interface for an operation having two arguments, the first being an integer, the second a character, and which returns an integer result. The use of **funct** indicates that no side effects are performed by the operation. If **proc** had been used, side effects would be expected. The ⟨identifier⟩'s can be optionally specified to assist in specifying either the result or the body of the operation.

Notation 2 A type definition is described syntactically as follows:

⟨identifier⟩: **type**
  representation:⟨type⟩;
  operations:
    {⟨identifier⟩ = ⟨operation interface⟩
      [⟨operation body⟩]}*
  end

For types that are primitive, the representation type is omitted, as are the bodies of the operations.

Note: Strictly speaking, identifiers are artifacts present in the character string form of the program that do not exist at the machine interface. Rather, operations are "named" by one of the primitive types, *e.g.*, bit_32, present at the interface. But, as in the use of assembly language where symbolic forms are used instead of bit patterns, we use identifiers instead of these primitive data values, with the understanding that the identifiers are replaced by the bit_32 values when the program is "assembled."

Notation 3 When we wish to denote the result of executing an operation, as opposed to the expression that generates the result, a dot(.) notation is used. Thus, plus(3,4) yields a result plus.(3,4) that happens to be identical to (*i.e.*, it denotes) 7.

We now introduce the examples. We begin with the definition of a primitive type expressed in the same form as a user-defined extension. Then, the extension mechanism is used to define an integer type, a type that is usually provided by the system. Finally, the primitive type constructor **union** is used to construct new types. Once again, the new type, though it cannot be user-defined, is nonetheless described in exactly the same form as a user definition.

```
bit_32: type
  representation: . . . primitive;
  operations:
   fixed_add = λ(bit_32,bit_32)funct(bit_32);
   fixed_sub = . . .; fixed_mult = . . .; fixed_div = . . .; . . .
   float_add = λ(bit_32,bit_32)funct(bit_32);
   float_sub = . . .; float_mult = . . .; float_div = . . .; . . .
   equal = λ(bit_32,bit_32)funct(bit);
   gtr_than = . . .; less_than = . . .; less_eq = . . .; . . .
   shift_right = λ(bit_32,bit_32)funct(bit_32);
   shift_left = . . .; . . .
   and = λ(bit_32,bit_32)funct(bit_32);
   or = . . .; x_or = . . .; . . .
   complement = λ(bit_32)funct(bit_32);
   zeros = funct(bit_32);
   one = funct(bit_32);
   . . .
   end bit_32;
```

Figure 1 The type definition for 32 bit values.

• *Example 1: a primitive type*

Our example is the "bit_32" type, specifying a "string" of 32 bits, but it is not the only primitive type of our machine interface. Perhaps other sizes of bits are also needed as types. Further, not all primitive types or aggregates will have their mappings to bits exposed at the interface. Indeed, it does not appear to be possible to expose all these mappings without opening the interface to potential compromise. This is particularly true for reference types, whose values must be scrupulously protected from arbitrary data processing operations.

In existing machines, of course, all data are ultimately expressible in bits. In realizing our machine interface, this mapping to bits is also made. However, users will not have access to, nor need they have knowledge of, this mapping except in the most general terms, *e.g.*, how expensive are various values and their operations in space and time. As a conventional machine does not expose its logic gates at its interface, some primitive data representations are not exposed in our type machine. However, by and large, all "ordinary" data are reducible to some form of bits.

The operations on "bit_32" values are those computational operations of the underlying hardware. The definition in Fig. 1 is meant to be indicative of this set of operations, but the list should not be considered to be the only possible one. Only the interface specifications for these operations are given as the operators are all primitives. This is the case for all the types with primitive operations. Note that both fixed- and floating-point arithmetic operations work on "bit_32" values. These are subsequently separately used to define integers and reals. The underlying machine does not, however, distinguish between these types. All its arithmetic operations work indiscriminately on bits.

```
int: type
 representation: bit_32;
 operations:
  add = λ(x:int,y:int)funct(int);
          on fixed_overflow then call integer_error(x,y);
          return(abs(bit_32.(fixed_add)(rep(x),rep(y))));
          end add;
  sub = . . .; mult = . . .; div = . . .; . . .
  equal = λ(x:int,y:int)funct(bit);
          return(bit_32.(equal)(rep(x),rep(y)));
          end equal;
  gtr_than = . . .; less_than = . . .; . . .
  bits_to_int = λ(x:bit_32)funct(int);
          return(abs(x));
          end bits_to_int;
  int_to_bits = λ(x:int)funct(bit_32);
          return(rep(x));
          end int_to_bits;
 end int;
```

Figure 2 The type definition for integer values.

As with all types, an initial source of values is needed. Here we suggest two sources, "zeros" which provides 32 zero bit values, and "one" which provides 31 zeros followed by a one bit. With these initial sources and the operations given, all 2^{32} bit patterns can be generated. These bit_32 values are merely bit patterns that are moved around and manipulated. They can be copied into several locations, etc., but do not themselves consume storage, and no storage allocation has been specified. That is to say, they are values, not objects.

There are some forms of operations that are not included in the definition of bit_32 even though the underlying machine clearly possesses them.

1. No branching or program control operations are included. At our interface, program control operations are buried in the primitive operations and do not appear as data processing operations.
2. No I/O operations are present, again because these are not to be regarded as data processing operations. Some form of I/O capability, however, must be present. For on-line memory, it may be possible to conceal the I/O under a one-level store interface to the storage hierarchy. For so-called source/sink I/O for communicating outside of the system, explicit operations of some form are required. These are not discussed here.

• *Example 2: a user-defined type*

What we describe in this example are integer values and not storage objects (cells) that can contain integer values. Thus, the operations that must be provided with integers are the usual arithmetic and comparison operations and two type conversion operations. Operations for updating and taking the contents of integer cells are not appropriate operations for integer values. They are appropriate for cells, which are described in the next section. The type definition for integer values is given in Fig. 2.

Only the body of one computational operation is shown, *i.e.*, that for "add." The other arithmetic operations are all similar. In "add," "fixed_add" is the primitive operation of the bit_32 type. It produces a bit_32 result if it returns normally. Otherwise, it calls the exception handler "fixed_overflow." This exception handler is defined in the "add" routine to invoke another exception handler for the integer type called "integer_error." This is a free variable that must be bound to some appropriate routine. We do not discuss this further here but see [16, 17]. If no exception occurs, the "add" operation returns the result of "fixed_add," converted by **abs** to an integer from a bit_32. The operation **abs** merely takes the same bit pattern and asserts that it is now to be regarded as an integer.

None of the arithmetic operations provides an initial source of integers. This source is provided by the "bits_to_int" operation which, given a bit_32, returns an integer. It is possible to be more restrictive. For example, perhaps only a source for the integer "one" might be given, all other integers being derived from it. This is needlessly primitive, however, and our goal is not minimality but convenience. No special construction method has been provided by the definition mechanism. Logic requires something analogous to "bits_to_int" but this operation is not treated specially. Several sources for integers could have been provided.

An operation, "int_to_bits," that converts integers to bit_32 values is also provided. This is analogous to the PL/I UNSPEC operation [18] and is not dangerous to the type system. Users must be cautioned that using it may make their programs representation-dependent, however. This operation is particularly useful for us when defining arrays. Primitive aggregates have bit_32 "selectors" so that, if we wish to use integer selectors instead, it is possible to easily convert the integers to these bit values.

No storage has been associated with integers. The integers are merely re-interpretations of the bit_32 values. Note that **abs** does not assert that the storage in which its argument value resides is to be treated as an integer. Rather, it returns a value completely unconnected with any implementation-required storage for its argument and asserts that this value (bit pattern) is an integer.

• *Example 3: unions and general variables*

Intuitively, one would like to treat union types as if they specified the set union of values of their constituent types

and indeed, this is the view we took in [7]. That view, however, creates a problem when we attempt to fit unions into the framework discussed in Section 2. Type conversions between union types and their constituent types are needed. We wish all computation, including such changes in representation as these trivial conversions, to be accomplished by explicit operations, and we need a source from which to acquire these operations, without exposing the representation of the union. The need for such conversions is particularly clear when it is remembered that we insist on an exact match between procedure parameter declarations and the arguments passed to them.

It is the union type that must provide these conversion operations, and aside from one testing operation, unions provide only conversion operations. Any computational operations are drawn from the constituent types. Since unions can contain a great, potentially unbounded, set of alternatives, we must provide a framework for coping with these possibilities. We present an example first and then discuss the way in which it is used. The operation **union** takes two existing types and produces the "union" type and hence is a type generator. We give an example in Fig. 3. Only operation interfaces are given as the operations are primitive.

Note that unions are examples of types that do not possess explicit construction operations, though by using "widen," it is possible to produce "union" values. To convert from some specific alternative type value to the corresponding union value, one must

1. Obtain the "widen" operation from the union as the result, *e.g.*, of

   union.(int,char)(widen).

2. Using the "widen" operation, which is of the form

   λ(**type** = T)**funct**(**funct-type** = λ(T)**funct**(**union.**(int,char)))

   when type argument T is set to int produces a specific "widening" conversion operation with interface

   λ(int)**funct**(**union.**(int,char)).

3. It is this specific widening function that can thus be used to convert from, *e.g.*, an int to the **union**.

A similar sequence is followed in order to convert from the union to a specific alternative type, only "narrow" is used instead of "widen."

This discussion of union types provides a natural context in which to discuss strong typing and type checking. Our view is that all type checking must be done at program construction time. It is important to point out that program construction can encompass more than compilation. It can include link editing as well, and in our machine interface, can even be performed during the execution of ordinary, user-written programs. It is the kind of checking which, *e.g.*, ensures that there is a type match between parameters and arguments, and between identifier usage and declaration. This view requires that types be known in order for a program to be constructed. This is our understanding of what strong typing means and implies that all type checking is "strong."

```
union (int,char):type
 representation: . . . primitive
 operations:
  narrow = λ(type=T)funct(funct-type = λ(union(int,char))funct(T));
  widen = λ(type=T)funct(funct-type = λ(T)funct(union.(int,char)));
  alternative = λ(union.(int,char))funct(type);
  end union.(int,char);
```

Figure 3 The type definition for union of integer and character values.

What happens with the "narrowing" of a union value is not type checking but type conversion. In this case, the success of the conversion depends on the value given it. Whether the value is appropriate is related to its "basic" type, but it could just as easily depend on some other property. For example, one might define a subrange of the integers as a type. Conversion from integers to the subrange type would depend on whether the integer value is within the desired subrange. This is not unlike a great many operations which return results only conditionally. The "add" operation of the integer type is one such operation.

The operation "alternative" yields the basic type of the value of the union. Exactly what constitutes a "basic" type is a troublesome notion. The simplest view is to consider any nonunion type as basic. This permits **union**(A,**union**(B,C)) and **union**(B,**union**(A,C)) to generate the same type. The result of "alternative" can be tested to provide alternative dependent computations, as in

if alternative(x) = char **then** call char_procedure(x);

No computational operations are provided with unions. The values of union type must be converted to their basic (nonunioned) types before such computations can be done. Neither updating nor contents operations are provided, since union types do not describe storage objects, but rather describe values. Of course, as we shall see, references can be generated to cells that can contain union values, but these belong to a different, distinct type.

```
cell.(int):template
    representation: . . . primitive
    ref operations:
        create = proc(ref.[int]);
        free = λ(ref.[int])proc;
        null = funct(ref.[int]);
        equal = λ(ref.[int],ref.[int])funct(bit);
        val = λ(ref.[int])funct(int);
        upd = λ(ref.[int],int)proc;
        end cell.(int);
```

Figure 4 The template definition for integer cells (reference operations only).

Discriminated unions might have been provided instead. To define them, all type arguments and results in the operations of the union type of Fig. 3 are replaced with tag field arguments and results. This is an advantage in languages in which types themselves are not values. In addition, the type generator **union** would require tag field values associated with each alternative of the union.

The **union** type generator can only produce finite unions of types. It is frequently useful to be able to specify infinite unions, *i.e.*, unions in which the number of basic types is unbounded. Such infinite unions may need to be supplied by means of primitives. However, we have no difficulty expressing the resultant type in terms of the operations upon it. A type **general**, which is a "union" over all possible types, can thus be described in the same way as more limited unions. The **general** type is useful for coping with so-called typeless languages. Of course, its operations "widen," "narrow," and "alternative" have a broader range of argument and return types, but the form of its type definition is otherwise like other unions. It should be emphasized that we are not introducing flexible objects. Rather, general values "include" any value that the type system will ever be able to produce.

4. Objects and templates

The purpose of objects in a high level language is to maintain state information over periods of a computation so that examination of this information can influence the subsequent course of the computation. The simplest objects we call cells. Cells permit values to be retained, via an update operation, then subsequently recalled. An aggregate object supports a restricted form of address computation, *i.e.*, that computation that yields a reference to one of its components. By carefully distinguishing values from objects, we avoid having update operations interfere with or affect this address computation. Specifically, flexible objects are not supported.

Objects are not directly accessible to operations of the machine interface. As in ALGOL 68, operations, indeed all routines, manipulate objects, *e.g.*, to change their state, by manipulating their references. All operations for data definitions then are associated with types. The operations that affect objects are associated with their reference types. This is what is meant by a value-oriented data definition facility. To our knowledge, this value-oriented approach is unique.

Templates are used to describe objects. Roughly speaking, templates describe the layout of storage. When we wish to deal with references to this storage, we apply the **ref** operator to a template, yielding a reference type whose operations manipulate storage of the form described by the template. As will become clear in the next section, templates can also be used to describe values that are structurally similar to the objects discussed here.

Templates are complicated by their subsequent use in deriving value types that are analogs of the objects. To cope with this, our syntactic form for a template provides not only for the specification of reference values but also for the specification of the derived values. Only the reference types are discussed in this section. The full syntactic form for templates is as follows.

Notation 4

```
⟨identifier⟩: template
    representation:⟨template⟩;
    ref operations:
        {⟨identifier⟩ = ⟨operation interface⟩
                        [⟨operation body⟩]}*
    value operations:
        {⟨identifier⟩ = ⟨operation interface⟩
                        [⟨operation body⟩]}*
    end ⟨identifier⟩;
```

The following examples present no single primitive storage object. Rather, what is primitive are operations that generate templates for storage objects when they are supplied arguments. For example, there is no primitive cell. Rather, **cell** is a primitive operator that, when given a type, generates a template for a cell that holds exactly the values of the type. Instead of primitive templates, the interface provides primitive template constructors.

• *Example 4: cell objects*

We illustrate integer cells to expose the differences between the integers of Example 2 and the cells that can contain them. Thus **cell**(int) is an expression that yields a template specifying cells that contain integers. To generate the associated reference type, one must then execute the **ref** operation with this template as its argument. The template for integer cells has its description given in Fig. 4. While **cell.**(int) could be used to denote templates for

cells of integers, we generally use a sugared form for cells in which square brackets enclose the cell's type. Thus

Notation 5 [⟨type⟩] denotes a cell capable of containing values of type ⟨type⟩. Further, **ref** is a unary operation with a template argument that is usually delimited by these square brackets. Because of this, the parentheses around its argument are frequently elided.

The representation template for **cell.**(int) is not presented since all objects generated using cell templates are primitive. There is no test that can be made by the user of the type which will reveal this representation.

The first four of the operations of the reference type for integer cells are operations that are common to all reference types. These operations permit the creation of the object ("create"), the destruction of the object ("free"), the generation of null reference values, *i.e.*, those reference values that do not refer to an object ("null"), and the testing of references for equality ("equal"). While every reference type has corresponding operations, these operations are always specific to the particular reference type. Thus, *e.g.*, the "create" operation above makes only integer cells. The "create" and "free" operations are procedures rather than functions, because they cause a "side effect" of altering the set of objects that are in existence. The "null" and "equal" operations are pure functions and are so designated.

The "val" and "upd" operations are specific to cell objects. That is, they perform the characteristic operations on cells of storing a value in a cell ("upd") and retrieving it ("val"). Storing a value produces a side effect, and hence "upd" is a procedure while "val" merely retrieves part of the state without altering it, and is a function. These are the basic operations involving the integer cell as a storage object. Note again that no arithmetic operations are given with integer cells. These operations belong to the type integer whose values can be contained in the cell. All cells, it should be noted, have analogous sets of operations, each having interfaces specific to the type of values that they contain.

No clue has been given as to how these primitive reference values are represented, *e.g.*, in bits. However, while we have not shown it, an operation "**ref**_to_bits" could be provided to convert reference values into the bit strings that represent them in the implementation. This operation might be useful for ordering references or for displaying them in some form. Such an operation does not violate the integrity of the typed interface, but it does introduce implementation dependency.

If instead of cells being primitive, a user were required to define them in terms of an underlying bit space object, then "val" and "upd" would be required to explicitly convert between integers and bit_32's. In "val," one could use the operation "bits_to_integer" to change a "bit_32" value into an "int" value. In "upd," the operation "int_to_bits" would be used to store the integer value's representation in the bit_32 cell. A serious difficulty with this alternative approach is the conversion from bits to a higher level type. This causes no problem for "int" and, indeed, for many types of values. But reference values present a special problem. If a conversion operation from bits to references is provided which simply asserts that some bit pattern is a reference, then no storage object is secure from inadvertent or malicious reinterpretation as some other object. Potentially, even free storage might be compromised. One way out, without going as far as to make **cell** primitive, as suggested in the example, is to make both cells for bits and cells for references primitive. That would at least make the storage system secure.

The advantage of **cell** being a primitive template generator is that the representation of types can remain completely concealed, and, in addition, the definition of references to cells of type T does not depend on the operations provided with type T. That is, there is no obligation on the part of the definer of type T to provide operations that enable its values (by being convertible to and from bits) to be stored in cells. They may always be so stored.

- *Example 5: structure objects*

Structures are storage objects possessing several named components, each of which is itself an object. Again, we emphasize that objects and values must be carefully distinguished. Here we only specify the operations associated with references to structured objects. Structured values which can be derived from the same template are also permitted and are treated in the next section.

We first introduce the following syntax for describing structures.

Notation 6

⟨template⟩::=[⟨identifier⟩:⟨template⟩
{;⟨identifier⟩:⟨template⟩}*]

Note here that the square brackets are not meta-language symbols but symbols of the language being described. They are used in the same way here as they are with cells. The example we will use is the following two-level structure:

Z = [x:[int];y:[a:[char];b:[int]]]

Then Z is defined in Fig. 5.

```
Z:template
    representation: . . . primitive
    ref operations:
        create = proc(ref.Z);
        free = λ(ref.Z)proc;
        null = funct(ref.Z);
        equal = λ(ref.Z,ref.Z)funct(bit);
        x = λ(ref.Z)funct(ref.[int]);
        y = λ(ref.Z)funct(ref.[a:[char];b:[int]]);
        end Z;
```

Figure 5 The template definition for structures described by Z (reference values only).

The role of a storage structure (aggregate) is to provide a controlled form of address arithmetic. In most languages the syntax at least implies that an object be in existence prior to any address computation. Thus, constructs such as "A[i]" or "Q.R.S." require the presence of the objects named "A" and "Q," respectively. Further, in SIMULA [13], the procedures of a class are part of an instance of the class, not part of the class definition. Hence, conceptually, all address computation is put off until run time, since only then do objects exist. Our approach splits the address computation into two parts. The "offset" computation depends only on the type and can be done at program construction time, while the conversion from "offset" to "pointer" must be deferred until the storage object exists, *i.e.*, at run time. The offset computation involves selecting from the type, the "offset function." Thus

ref.Z(y) → **ref.**Z(y)

where **ref.**Z.(y) is the function "y" in the definition of Fig. 5, takes a **ref.**(Z) value as an argument, and returns a **ref.**[a:[char];b:[int]] result.

For multi-level selection, one must compose functions. Consider

λ(P){**ref.**[a:[char];b:[int]].(a) (**ref.**Z.(y)(P))}

where P is a symbol denoting a **ref.**Z value. Here, **ref.**[a:[char];b:[int]].(a) denotes the function that takes a **ref.**[a:[char];b:[int]] argument and returns a **ref.**[char] result. When composed with the previous offset function, as occurs above, the resulting function denotes an offset function that takes a **ref.**Z argument and returns a **ref.**[char] result, a reference to component "a" in component "y" of the structure. Thus, selection operations, like any other operations, are extracted from the type definition. This requires, of course, that an instance (value) of the type be passed to each operation explicitly rather than there being an implicit association by means of the operator being a part of the value.

There are problems with competing forms of address computation. We consider two alternatives.

1. A general purpose **select** operation, *i.e.*,

   select (ref to object, name of component) → ref to component

   What is the interface specification for **select**? Since **select** must work on any reference and for any component, only imprecise type information can be put in its interface constraints. Thus, the type of its result is not such that an expression involving "select" can be readily incorporated into a larger expression unless it becomes an argument that is itself only very imprecisely stated. Further, no part of the address computation can be precomputed without the precomputation having access to an underlying machine and hence violating the interface. Why? Because a reference to the object must exist when **select** is executed.
2. Reynolds [19] has suggested treating the object itself as the selection function which, when given a component name, returns a reference to this component. Thus, if Q is a symbol denoting a reference to a structure, then "Q(A)" yields a reference to the "A" component of Q. However, this still leaves the result type imprecisely specified. Again, of course, Q is not available until run time.

A structure template, like a cell template, has the standard object related operations of "create," "free," "null," and "equal." Also, structure templates are primitive and have no revealed template representation.

- *Example 6: row objects*

A row template describes a storage object consisting of identical object components. This template is produced using the **row** operation which requires the row size and component template as arguments. We treat selection in rows somewhat differently from selection in structures for two reasons.

1. The components of a row are identical, and hence we wish to make information concerning the form of the components available as soon as possible.
2. "Address computation" is frequently performed at run time, *i.e.*, the desired component of a row is computed dynamically.

It is because of (1) that it is possible to conveniently handle the dynamic computation of components (2). Thus, in Fig. 6, we describe a template for a row consisting of 10 integer cells. As before, a notation is introduced for these templates so as to make the specification more readable.

Notation 7

[(⟨size⟩)⟨template⟩] shall denote a row of a given "size" whose components are described by the ⟨template⟩.

With structures, the component names appear explicitly as the names of "offset" functions in the reference type for the structure. With rows, a "select" function is provided that can compute the "offset" function at run time. Why? Because of the following:

1. The "select" function can be extracted from the type and provides explicit information as to the form of the components, *i.e.*, it returns an "offset" function that produces a **ref.**[int] when given a **ref.**([(10)[int]]); *i.e.*, its return type is

   λ(**ref.**([(10)[int]])**funct(ref.**[int])

   Thus we are given detailed type information concerning the component's type from the start.
2. The result of a bit_32 computation at run time may be used to determine the desired element. Using the select function itself at run time does not result in a loss of required type information. If the component names were in the "row" type directly, then we would lose information concerning the types of the components since we would then be required to use the row type as a function to obtain the offset function.

If we are presented with "A[i+j]" as an address computation, then at program construction time, the select function for A's type is extracted and the following code generated:

{**ref.**[(10)[int]].(select)}(i+j)(A)

where the braced part of the computation is done during program construction and yields the "select" function. Applying it to (i+j) produces the offset function, which can then be applied to the specific row A.

This organization permits the isolation and identification of common expressions. For example,

"A[i+j] ← B[i+j]"

could be represented by

(a) f ← **ref.**[(10)[int]].(**select**)(i+j);
(b) f (A) ← f(B);

where ← represents the "upd" operation. Step (a) has computed the offset function for both the arrays. The actual assignment of a component of B to a component of A at (b) does not require the computation of this offset function to be done twice.

If the subscript for an array reference is constant, then the offset function can be generated at compile time and embedded directly in the code. Thus "A[5]" could be represented by

{**ref.**[(10)[int]].(select).(5)}(A)

where the "offset" function itself is generated at compile time by the braced computation, leaving only the computation of the component reference from the aggregate reference.

```
row.(10,cell.(int)):template
    representation: . . . primitive
    ref operations:
        create = proc(ref.[(10)[int]]);
        free = λ(ref.(row.[(10)[int]])proc;
        null = funct(ref.[(10)[int]]);
        equal = λ(ref.[(10)[int]],ref.[(10)[int]])funct(bit);
        select = λ(bit_32)funct(λ(ref.[(10)[int]])funct(ref.[int]));
        end row.(10,cell.(int));
```

Figure 6 The template definition for rows of ten integer cells (reference operations only).

As in previous examples, the representation for a row is primitive. Further, the operations "create," "free," "null," and "equal" are as for other references to objects.

- *Example 7: dynamic row objects*

The rows of Example 6 are all such that their size must be known at program construction time. This is similar to the Pascal restriction [20] which has been the subject of some controversy. Most languages, starting with ALGOL 60, provide more dynamic sizes. There are three generalizations that we can imagine with respect to the size variability of rows.

1. A parameter may have a type such that it is compatible with rows of a given component type but with an unspecified (or incompletely specified) size. Thus, any reference to a fixed size row could be converted to this form of reference to rows of unknown size. This extension does not require any new template at all but merely a new type construction capability for infinite unions of existing types, in this case involving references to various size rows. No new template is needed because all storage allocation is still being done using row templates specifying a fixed size. The operations of such a union are exactly those we have seen in Example 3.
2. Every instance of a row is still of fixed size, but this fixed size need not be specified until the time that an instance of the row is to be created. Then, the size is given as an argument to the "create" operation. This kind of row of settable size is supported by a diversity of languages, including ALGOL 60 and PL/1. This form

```
d_row.(cell.(int)):template
 representation: . . . primitive
 ref operations:
  create = λ(bit_32)proc(ref.[(*)[int]]);
  free = λ(ref.[(*)[int]])proc;
  null = funct(ref.[(*)[int]]);
  equal = λ(ref.[(*)[int]],ref.[(*)[int]])funct(bit);
  select = λ(bit_32)funct(λ(ref.[(*)[int]])funct(ref.[int]));
  narrow = λ(type=T) funct (funct-type = λ([(*)[int]])funct(T));
  widen = λ(type=T) funct (funct-type = λ(T)funct([(*)[int]]));
  size = λ([(*)[int]])funct(bit_32);
  end d_row.(cell.(int));
```

Figure 7 The template definition for a dynamic size row of integer cells (reference operations only).

does require a new template constructor in which the resulting create operation is parameterized with respect to the desired size of the row. This is the case we treat here.

3. Row size is completely dynamic such that operations subsequent to object creation can alter its size. A number of subcases might be considered under this, such as (a) the number of components is permitted only to grow; (b) components can disappear but once gone cannot reappear; (c) components can freely come and go. The last case is a version of our nemesis "flexible objects." This is the form of object that we have been trying hard to avoid and hence is not supported.

The definition of a dynamic size row [case (2)] then is given in Fig. 7. Syntactically, dynamic size rows are denoted by an "*" in the size position of the denotation for other rows. The **d_row** operator constructs the template given the component template as argument.

The operations "create," "free," "null," and "equal" are the basic ones associated with all reference types. Note that here the "create" operation takes an argument which specifies the size of the dynamic row. The "select" operation has the same interface as its counterpart with fixed size rows and is used the same way. Importantly, however, the subscript range checking previously performed by the "select" operation must now be performed by its resulting offset function at run time.

The operations of "widen" and "narrow" are analogous to their like-named counterparts in the definition of unions. This reflects the fact that references to dynamic rows can also refer to fixed size rows. Thus, references to dynamic rows can provide the function mentioned in case (1) above, that of specifying as parameters rows whose sizes are unknown. Because we wish this to be the case, some care must be taken in how references to dynamic rows are represented. At the interface, the representation is concealed, but one must be selected for the implementation. In order to cope with the union-like attributes of dynamic rows, this representation must involve an indirection; *i.e.*, dynamic rows cannot have a contiguous representation where the size is stored adjacent to the elements of the row. Such a representation would make the implementation of the "widen" operation impossible. Its representation might then consist of a pair, ⟨size of row, address of its storage⟩. This is what has traditionally been called a "dope vector."

The operation "size" replaces the analogous operation "alternative" in unions. Instead of yielding a type value that can be tested, as done by "alternative," "size" yields a bit_32 value that indicates the number of elements of the dynamic row. Thus, this operation need merely access the dope vector. The size information is more useful because it can be passed directly to a subsequent "create" operation, thus enabling more like-sized rows to be created.

5. Values derived from objects

Section 1 mentioned a duality between objects and values, but we have not seen this as yet. In this section, we illustrate how templates can be used not just to describe objects, and hence their references, but also values which share a common description with the objects. The motivation for this is to obtain aggregate values. Three questions come to mind.

Why are aggregate values of interest?

Pure value semantics is desirable for both theoretical and practical reasons. Knowing when side effects can occur and, more importantly, when they cannot, is a great aid in analyzing and understanding the program. Read-only references do not have that desirable property, since they merely ensure that the user of the reference cannot modify an object by means of the reference, not that there is no way of modifying the object. The "immutable objects" of object-oriented storage models, though adequate, give the wrong connotation, implying separate storage even for the smallest immutable object (though optimization can sometimes eliminate it).

Related to value semantics is the desirability of providing an atomic update operation for aggregates in their entirety. With aggregate values, this becomes simply an ordinary scalar update, perhaps handled via hidden indirections, but which can readily be assured of atomicity, since no operation can change a value.

Finally, unions of types whose values are aggregates yield "flexible" aggregates without introducing flexible objects. Cells whose values are such unions are freely

permitted. Since references to components of these aggregate values are not possible, the disappearance and reappearance of various aggregate components have no pernicious effects.

Why make these aggregate values via templates?
If aggregate values are desirable, then their types must be formed in some way. Making them via templates emphasizes their structural correspondence with the object form of aggregate. Of course, having the same template describe both provides a certain economy at the interface.

Perhaps most important is that aggregate objects provide a convenient way of generating aggregate values. Rather than have an entire other set of operations for constructing aggregate values, one component at a time, we envisage the existence of one additional operation with aggregate values, the "enclose" operation. This operation takes a reference to an object as an argument and produces the value form of the object. Thus, producing aggregate values is accomplished by allocating an aggregate object, initializing its components, and then "enclosing" the resulting object to yield the value. There is a potentially large computational economy here. Value transformers must conceptually make new copies for each transformed value, rather than do the modification in place. While it may frequently be possible to avoid these copies, extra implementation complexity is needed in order to do so. Enclosing of the object form of aggregate after it has been updated in place provides explicitly the best that can be achieved.

While this view is quite useful, it is not essential. It is possible to conceive of primitive type generators that produce types describing aggregate values and providing operations to construct these values. Thus, it should be kept in mind that how aggregate values or their types are generated is independent of the desirability of having them.

Why not handle all values this way?
There are values that currently have no analogous object form. These fall into three categories.

1. The structure of the template definition provides no way of representing a reference. The template describes the storage object, not its reference, whose representation is entirely concealed. It is essential, of course, that references be very tightly controlled so as to protect the type system's integrity.
2. To complete the machine interface, values for program material, *e.g.*, procedure and function values, need to be included. It is unclear how these can be regarded as storage objects.
3. Some values simply do not warrant an object form. Integers are such values, and it is useful to be able to describe them without recourse to storage objects.

```
cell.(int):template
    representation: . . . primitive
    value operations:
        val = λ(value.[int])funct(int);
        enclose = λ(ref.[int])funct(value.[int]);
        end cell.(int);
```

Figure 8 The template definition for integer cells (value operations only).

The liability of the approach is the complexity introduced so as to permit value forms for aggregate objects when they may have no such useful form. While there is no requirement to provide value operators, one is still left with a template definition structure that is more complicated than that required solely for objects. We return to the subject of templates, their formation, and their features, in Sections 6 and 7.

We now proceed to the examples. These object-derived values complete the definitions of the template of the last section. That is, the value examples are for integer cells, structures, etc. In all the examples, the operations of the reference values are omitted in the same way that in the previous section the value operations were omitted.

- *Example 8: cell values*

This simplest example reveals much that is universal for all the primitive template generators. The object-oriented operations of the template, *i.e.*, those involved in creating and destroying objects and those peculiar to references such as "null" and equality of references, are not operations of the type that describes object-derived values. Those involved in modifying objects or which permit the subsequent modification of objects are either not present or are transformed so as to rule out in-place modification.

The value form of integer cells is given in Fig. 8. This completes the specification of the cell template. The only operations are "val," the contents-accessing function, transformed now so as to work on the value form of cell, and "enclose," the operation that converts the object form of cell to the value form. An "enclose" operation is present in all definitions of values derived from templates. Indeed, it is the reason why this approach has been pursued. It is an operation of the template that in some sense belongs to both the value and the reference types. We place it in the value specification here and in the subsequent examples because it is the existence of the value forms that requires its presence. More is said about "enclose" in Section 7.

```
Z:template
    representation: . . . primitive
    value operations:
        x = λ(value.Z)funct(value.[int]);
        y = λ(value.Z)funct(value.[a:[char]; b:[int]]);
        enclose = λ(ref.Z)funct(value.Z);
        end Z;
```

Figure 9 The template definition for the structure Z (value operations only).

```
d_row.(cell.(int)):template
 representation: . . . primitive
 value operations:
  select = λ(bit_32)funct(λ(value.[(*)[int]])funct(value.[int]));
  narrow = λ(type=T) funct (funct-type = λ(value.[(*)[int]])funct(T));
  widen = λ(type=T) funct (funct-type = λ(T)funct (value.[(*)[int]]));
  size = λ([(*)[int]])funct(bit_32);
  enclose = λ(ref.[(*)[int]])funct(value.[(*)[int]]);
  end d_row.(cell .(int));
```

Figure 10 The template definition for dynamic size rows of integer cells (value operations only).

• *Example 9: structure values*

As with cells, we complete the specification of the structure template whose reference operations were given in a previous example. Thus, we use the structure of Example 5, *i.e.*, Z = [x:[int];y:[a:[char];b:[int]]]. The value operations for template Z are given in Fig. 9. As with cells, the operations "create," "free," "equal," and "null" do not appear as value operations. The selector operations have been modified so as to work on values of form Z and to return the value form of the components as results. Thus, because references are not returned, no subsequent operations result in the modification of the value. The implementation may, for values requiring large amounts of storage, use indirect addressing, thus sharing this storage among all cells containing the same value. On the other hand, for small values, copying the entire representation is both feasible and frequently desirable. No operation is present that can distinguish between these approaches.

As with cells, an "enclose" operation is present to provide convenient initialization of the object-derived values. With aggregates, this is particularly important. While value forms of cells are included only for completeness, easy creation of aggregate values is the purpose of the separate value and reference types with their "enclose" operation. Flexible types specifying many aggregate forms are achieved by taking a union of types that describe aggregate values. Flexible locations cannot be produced.

• *Example 10: row values*

In Section 4, two kinds of rows were treated, those whose type determined the size of the objects and those whose size was specified at creation time, which we called dynamic rows or d_rows. Both have value forms. Here, only d_rows are treated. Fixed size rows are a restricted case of d_rows with only two of the operations supported.

The specification for the value form of d_rows of integers is given in Fig. 10. Notice again that no object-oriented operations appear in its definition. The operations remaining are simply the "union"-oriented ones for providing conversions to and from fixed rows and for identifying the current row's type (size), plus the selection operation. These have all been modified to operate on d_row values. For fixed rows, only the "select" operation among these would remain. In both cases, an "enclose" operation is included to permit initialization of row values.

A cell containing d_row values provides the analog of the ALGOL 68 flexible rows, which in ALGOL 68 result in flexible objects whose components can appear and disappear based on the currently assigned row. With d_row values, however, it is clear that references to components are not permitted since the d_rows are not objects but are values whose components cannot be referenced. The selection of a component of a value produces the component's value as a result, not a reference. These d_row values then provide a justification for the ALGOL 68 restriction that references to components of flexible rows are not permitted. Rather than simply being an *ad hoc* restriction due to implementation problems, however, the restriction can be seen as an intrinsic requirement of the object/value distinction that we have been making.

6. User-defined templates

The examples of the last two sections dealt only with the templates produced by the primitive template-generating operations. In this section, we define a template using the extension mechanism provided to the users. The example is that of strings. While it is not remarkable in its sophistication, it illustrates the flexibility and uniformity of the extension mechanism.

• *Example 11: strings*

Using strings as an example is interesting for a number of reasons. It is not normally chosen to demonstrate a data definition mechanism. Strings are frequently provided as primitives, and hence have not been user-defined. Further, there is some confusion about whether strings are objects or values. For instance, IBM PL/I [18] does not permit aggregates (objects) to be returned by procedures, but strings may be returned, hence treating strings as values. In addition, however, part of a string may be modified in place by means of the "SUBSTR" pseudovariable,

which treats the strings as objects. Using a template definition, strings are defined here in both object and value form. Unlike PL/I, however, our definition results in the value and object forms being distinct. Thus, updating in place is appropriate for string objects but not string values, while returning strings is appropriate for string values but not for string objects. Figure 11 contains the definition of character strings. Part (a) defines the operations on references to character string objects, while part (b) defines the operations on character string values.

A char_string object has the normal operations for objects, *i.e.*, "create," "free," "null," and "equal." Here, however, they must all be user-specified. This is very simply done using the corresponding **ref.**[(*)[char]] operations and making the appropriate **rep** and **abs** conversions. The body of the "create" operation shows how this is done. Only the interface specifications are given for the rest of these operations.

The only new operation is "substr," which permits multiple components of a string to be updated with a single operation. The body of "substr" reveals how this is accomplished by updating each component character cell in turn. A select operation is also provided to permit access to the values contained in the char_string object.

The char_string values are more interesting. No object-oriented operations are associated with them. However, those operations normally considered to be string operations are all supplied. These include the string value comparisons of "equal," "less," and "greater," and the string manipulation operations of "substr" and "concatenate," as well as provision for empty strings with the operator "empty." The body of "concatenate" illustrates how new string values are constructed via the allocation and enclosing of objects. The method used by the "substr" operation is similar so we show only its interface.

What we have succeeded in doing is to provide the function of PL/I strings via the value and object forms of strings defined here. Importantly, these functions have not been provided through a single type but rather by means of the two types of strings. Values and objects have been strictly distinguished, avoiding both confusion and flexible objects.

```
char_string:template
  representation: d_row.(cell.(char));
  ref operations:
    create = λ(x:int)proc(ref.char_string);
             bit_32:t = bit_32.(int_to_bits)(x);
             return(abs(ref.[(*)[char]].(create)(t)));
             end create;
    free = λ(ref.char_string)proc;
    null = funct(ref.char_string);
    equal = λ(ref.char_string,ref.char_string)funct(bit);
    substr = λ(target:ref.char_string,start:int,len:int,
             source:ref.char_string)proc;
             ref.[(*)[char]]:t = rep(target);
             ref.[(*)[char]]:s = rep(source);
             bit_32:first = bit_32.(int_to_bits)(start);
             bit_32:number = bit_32.(int_to_bits)(len);
             bit_32:i;
             λ(bit_32)funct(ref.[char]):f;
             λ(bit_32)funct(ref.[char]):g;
             do i ← first to bit_32.(fixed_add)(first,number)-1;
                 f ← [(*)[char]].(select)(i);
                 g ← [(*)[char]].(select)(i+1-first);
                 f(t) ← g(s);
                 end;
             end substr;
    select = λ(int)funct(λ(ref.char_string)funct(ref.[char]));
    end char_string;
```

(a)

```
char_string:template
  representation: d_row.(cell.(char));
  value operations:
    equal = λ(value.char_string,value.char_string)funct(bit);
    less = . . .; greater = . . .;
    substr = λ(target:value.char_string,start:int,len:int)
             funct(value.char_string);
    concatenate = λ(a:value.char_string,b:value.char_string)funct
                  (value.char_string);
             value.[(*)[char]]:x = rep(a);
             value.[(*)[char]]:y = rep(b);
             ref.[(*)[char]]:z = ref.[(*)[char]].(create)(size(x)+size(y));
             do i ← 1 to value.[(*)[char]].size(x);
                 z[i] ← x[i];
                 end;
             do i ← 1 to value.[(*)[char]].size(y);
                 z[i+size(x)] ← y[i];
                 end;
             return(abs(enclose(z)));
             end concatenate;
    empty = funct(value.char_string);
    enclose = λ(ref.char_string)funct(value.char_string);
    end char_string;
```

(b)

Figure 11 *The template definition for character strings:* (a) reference operations only and (b) value operations only.

7. More about templates

A separate syntax for templates was provided in Section 4, which showed templates as consisting of three parts: a representation template, a set of reference operations, and a set of value operations. We emphasize this three-part form because templates themselves will, in our machine interface, have to be generated by explicit operations. Here, we suggest that the formation of a template requires an operation that takes two arguments, a type definition for reference values and a type definition for the corresponding enclosed values. The representation template can be deduced from the representations used for the reference and value type definitions. Thus, for the reference type, the representing type must be a reference type that refers to an object described by a template. This can be regarded as the representation template. Similarly,

the value type has a representation derived from a template, and these two representing templates must be the same. With this view, we have reduced the number of ways that types can be defined by users to one, *i.e.*, via a type definition. Templates act as repositories for these two type definitions.

Let us reconsider the role of templates. They do more than just bring together the two constituents above. Their fundamental role is to prescribe exactly the limits of the objects they describe. That is, a template specifies precisely what is to be considered as part of an object *versus* what is merely referenced by it. Templates play the same role with the corresponding object-derived values. Thus, when an object is enclosed via an "enclose" operation, yielding the value form of the object, the template determines what is considered as part of the value *versus* what is referenced by it. Only that which is part of the object becomes part of the corresponding value.

Templates can, of course, be included in larger templates. This permits objects to contain components that are themselves objects, where these components are part of the larger object, not referenced by it. It is this form of composite object that, *e.g.*, the CLU object-oriented model does not provide. It is also this form of composite object that, in order to avoid flexible objects, necessitates the distinction between templates and types. Unions of types are permitted, but not unions of templates. If all components were merely referenced by an aggregate, then referencing an object's component would yield a reference to a separate object. Changing the characteristics of this component would not affect the references to this free standing (former) component that has now been replaced by a component of a different form. With the component directly included in the major aggregate, however, a change in a flexible component affects previous references that continue to refer to the component in its changed form.

Another role for templates is that of specifying properties for the data they describe when those data are components of larger aggregates. These properties are initialization and enclosing. Since they are associated with the template, the reference and value types remain the same.

- *Initialization*

Since there is no convention requiring specific operations and, in particular, no "create" operation need be specified in either reference or value types, the initialization of a component object becomes a problem. How this initialization is accomplished should not be considered as helping determine the types, either reference or value, but is solely an attribute of the template.

How then does a "create" operation for the larger template specify the initialization for its components when those components require particular initial states? Either (1) it must be able to "see" the uninitialized states so as to perform the initialization itself, or (2) it must, by some convention, be able to invoke appropriate operators of the components to be initialized. In case (1), it has explicit access to the uninitialized state and may be able to subvert the intention of the definer of the component. In case (2), additional requirements must be placed on template definitions so as to provide this capability while preventing the use of the type operations prior to initialization. It is this second approach, with its assurance of type integrity, that is pursued. What is needed then is a general method by which the definer of the template can associate initialization with a template no matter how it is used.

Any template τ can be augmented with an initialization specification. This is in the form of a procedure with an argument of type **ref.**(τ) which, by means of side effects on its argument, performs the initialization. This initialization specification is associated with a template by means of the **init** function with the following interface.

init = λ(τ:**template-type**,g:**proc-type**)**proc**(τ initialized by g = λ(ref.τ)**proc**))

Thus, initialization need not be built into the template (nor into the operations of "ref" and "value" types) as a "primitive" operation but can be varied to suit the user's purposes. And no special naming convention for initialization operations is required, since the initialization operations are not part of either **ref** or **value** types. What is required of the types, of course, are primitives that permit the initialization procedures to be written. In the absence of explicit initialization, a template is initialized as specified by its representation template, if any, or by the initialization specified for its various components separately.

While the initialization associated with a template does not play a role in the associated type definitions, the initialization associated with the templates of the representations for reference and value types does. Thus, *e.g.*, a reference type is determined not only by its operations but also by the initialization of its representing type's template.

When providing initialization for aggregates constructed using the built-in template generators, it seems reasonable to provide it in two forms. Initialization for the entire aggregate can be provided as before. Alternatively, if the aggregate template was generated using component templates that already had initialization specifications,

then, in the absence of explicit initialization for the entire aggregate, these component initializations can be used. Let us illustrate this initialization with our three built-in template forms.

1. *Cells* In the absence of initialization, a cell template, *e.g.*, for int, produces, when the cell is formed, an uninitialized cell, perhaps with an undefined value, perhaps with a default value, or perhaps with no value, in which case an exception would be raised if its contents were requested. We can provide for the initialization of a cell via the primitives defined for **ref.**[int]. Thus

   τ = **init**([int],λ(x:**ref.**(int))**proc; x ← 25; end;**)

   yields a template which, when used in specifying a larger aggregate, describes an integer cell component that is initialized to the value of 25. This is true for both storage objects and enclosed (aggregate) values.
2. *Rows* Without initialization, a **row** template, *e.g.*, for [(10)[τ]], produces a row in which each element is initialized using whatever initialization is associated with τ. If, for example, the τ of (1) above were used as the [int] template for such a row, then all ten elements of the row would be integer cells initialized to the same value, *i.e.*, 25.

   When explicit initialization is provided, it overrides any component initialization. Thus

   init([(10)[int]],λ(x:**ref.**[(10)[int]])**proc**
   do i ← 1 to 10; x[i] ← i; **end**;)

   produces a row in which the first element contains one, the second element contains two, · · ·, and the tenth element contains ten. Again, this is the case both for storage objects and enclosed (aggregate) values.
3. *Structures* The situation with structures is similar to that for rows, but each component may have a separate initialization. Thus, we might have

   Z = [x:**init.**([int],λ(x:**ref.**[int])**proc; x ← 25; end;**),
   y:init.(
   [a:**init.**([char],λ(x:**ref.**[char])**proc;** x ← 'q'; **end**);
   b:[int]],
   λ(x:**ref.**[a:[char],b:[int]])**proc;**
   x.a←'r'; x.b←4; **end;**)]

Here Z has no initialization and so inherits, by default, the initialization of its components. Component x, an integer cell, is initialized to 25. Component y is itself a structure with explicit initialization. While y.a has explicit initialization (y.b does not), it is overridden by the initialization for y which sets y.a to 'r' and y.b to 4.

- *Enclosing*

The other capability provided by the template is the ability to enclose an object form so as to yield its value form. Notice, since the "enclose" operation relates reference values to the value form of the object referenced, that when types are defined independently, there is no way for a user to provide this operation in the reference and value types he defines. Such an enclose operation would, in order to be programmed, need an operation in the reference type that exposes the representing type so that the "enclose" operation of the representing type can be used to obtain the value form of the representation. Then, the **abs** function can be applied to produce the user-defined value. But the exposure of the representation should be discouraged. Indeed, the intent of the user-defined types is to hide this representation.

Our solution to providing an "enclose" operation is to associate it with the template, rather than with either of the types. This has the virtue, in addition, of not changing the constituent types of the template. Thus, an **enclose** operation is defined that has a template argument and that returns an "enclose" operation that is specific to the types of the template. Thus, no type checking information is lost when using this specific "enclose" operation. In the examples, the "enclose" operation was always associated with the value type. This was an expedient so as to simplify the presentation. Our real intent is that the reference and value types be unaffected by their being included in a template, and the solution just presented has this desirable property.

8. Discussion

The value-oriented view presented here is yet another of our efforts to define a storage model for programming languages and to integrate such a model into our operation-oriented machine interface. It is a considerable advance over our last such effort [7]. While the object/value distinction has been retained, the new view of type presented here smoothly integrates the storage model with a type definition facility. Clearly, our operation-oriented view has also been furthered, as can be seen readily from the operational nature of the type definitions presented.

The new value-oriented type definition mechanism provides a unifying framework. Unlike, *e.g.*, the new DOD language ADA [1], it has not been necessary to complicate the picture with new notions such as "sub-type" when sub-ranges are desired. Rather, a sub-range, *e.g.*, the integers from one to ten, is merely another type. Like a union type, it need only provide conversion operations between it and the integers. All computational operations can continue to be performed on integers. At a syntactic level, it may be desirable to provide a way of implicitly

specifying these conversions. We are convinced that this is a useful way of viewing subranges.

While progress has been made, the complete definition of the machine interface that is our goal requires much additional effort. The construction and manipulation of types and program material have barely been touched on. The operations required in this area should enable programs and types to be incrementally constructed in small pieces. Thus, fragments of these values must also be values. In particular, type and program material with free symbols must be manipulatable values. Operations to bind and resolve symbols must be provided. This is an effort that has never really been carried very far before. We have previously worked toward this [16, 17], and while many of the notions presented there are worth pursuing, those efforts were incomplete and did not avail themselves fully of the value-oriented view presented here. Probably the largest stumbling block is that fixpoint operations are needed in this area to construct both recursive programs and recursive data structures.

Polymorphic operations and parameterized types can both be provided by using these program and type construction operations. Polymorphic operations are realized using a program construction program. Such a program is given one or more type arguments and produces a result which is a program. The resultant program has arguments and result types taken from the types supplied as arguments. Parameterized types can be handled by type construction programs in a similar way.

Acknowledgments

This paper is a much revised form of an earlier memo written while the author was on sabbatical at the University of Newcastle-upon-Tyne. The encouragement and support of Professor Brian Randell is gratefully acknowledged. Dan Berry provided comments on an intermediate version that resulted in an improved organization for this final paper. This work was partially supported by Professor Randell's grant from the Science Research Council of Great Britain.

References

1. J. D. Ichbiah, "Preliminary ADA Reference Manual," *Sigplan Notices* **14,** Part A (1979).
2. B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," *Commun. ACM* **20,** 564–576 (1977).
3. B. Wegbreit, "The Treatment of Data Types in EL1," *Commun. ACM* **17,** 251–264 (1974).
4. N. Wirth, "Modula: A Language for Modular Multiprogramming," *Software Pract. Exper.* **7,** 3–35 (1977).
5. *An Informal Definition of Alphard, A Preliminary Report CMU-CS-78-105 of Carnegie-Mellon University*, W. A. Wulf, Ed., Pittsburgh, PA, 1978.
6. D. B. Lomet, "A Cellular Storage Model for Programming Language," *Research Report RC4360*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1973.
7. D. B. Lomet, "Objects and Values: The Basis of a Storage Model for Procedural Languages," *IBM J. Res. Develop.* **20,** 157–167 (1976).
8. A. Demers, J. Donahue, and G. Skinner, "Data Types as Values: Polymorphism, Typechecking, Encapsulation," *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, Tucson, AZ, January, 1978, pp. 23–30.
9. J. Mitchell and B. Wegbreit, "Schemes: A High Level Data Structuring Concept," *Current Trends in Programming Methodologies*, R. Yeh, Ed., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.
10. J. McCarthy *et al., LISP 1.5 Programmer's Manual*, MIT Press, Cambridge, MA, 1966.
11. "Revised Report on the Algorithmic Language ALGOL 60," P. Naur, Ed., *Commun. ACM* **6,** 1–17 (1963).
12. B. Liskov and S. Zilles, "Programming with Abstract Data Types," *SIGPLAN Notices* **9,** 50–59 (1974).
13. O.-J. Dahl, B. Myhrhaug, and K. Nygaard, *The SIMULA 67 Common Base Language*, Publication S-22, Norwegian Computing Centre, Oslo, 1970.
14. C. H. Lindsey and S. G. van der Meulen, *Informal Introduction to ALGOL 68*, North-Holland Publishing Co., Amsterdam, 1971.
15. A. Van Wijngaarden, B. J. Mailoux, J. E. L. Peck, and C. H. A. Koster, "Revised Report on the Algorithmic Language ALGOL 68," *Acta Informatica* **5,** 1–236 (1975).
16. D. B. Lomet, "An Operator Driven Model of Program Execution," *Research Report RC4444*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1973.
17. D. B. Lomet, "Control Structures and the RETURN Statement," *Information Processing '74*, North-Holland Publishing Co., Amsterdam, 1974.
18. *OS PL/1 Checkout and Optimizing Compilers: Language Reference Manual*, Order No. GC 33-0009-4, available through the local IBM branch office.
19. J. C. Reynolds, "GEDANKEN—A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept," *Commun. ACM* **23,** 308–319 (1970).
20. N. Wirth, "The Programming Language Pascal," *Acta Informatica* **1,** 35–63 (1971).

Received February 12, 1980; revised June 20, 1980

The author is located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.