On the Use of Abstract Data Types to Specify the Modules of a System*

Øystein Helle

TR 81-469
October 1981

Department of Computer Science
Cornell University
Ithaca, New York 14853

permanent address:
Institute of Mathematics
University of Bergen
N-5014 Bergen-U
Norway

*The note was written during a one-year visit financed in part by the Norwegian Research Council for Arts and Sciences during 1981.

I would like to thank Professor David Gries for his helpful suggestions and his excellent proofreading.

CONTENTS

INTRODUCTION.

In order to construct a large and complex system, it is necessary to decompose it into smaller parts. Specifying the properties of these constituent parts or modules of a system is a major problem in informatics.

The notion of a module is vague and ambigious and to clarify my use of the word, I quote the papers [Parnas, 72] and [Winograd, 79]:

Parnas:

> "A "module" is considered to be a responsibility assignment rather than a subprogram. The modularizations include the design decisions which must be made before the work on independent modules can begin."
>
> "The order in time in which processing is expected to take place should not be used in making the decomposition into modules."

Winograd:

> "The building blocks out of which systems are built are not at the level of programming language constructs. They are "subsystems" or "packages", each of which is an integrated collection of data structures, programs and protocols."
>
> "We need to shift our attention away from the detailed specifications of algorithms, towards the description of packages and objects with which we build."

Hopefully, it is clear from these quotes that I am interested in the specification rather than the implementation of modules. Furthermore, I consider a module to be the definition of a data structure that can represent some information together with a number of functions or programs that provides the means by which a user can access the data in the structure. The implementation of a module should be invisible to the user, who needs to know only the specified properties of the functions and programs.

There are obvious similarities between this notion of a module and the concept of an abstract data type. An abstract data type defines a data structure and the functions that can create and operate on instances of the structure. My opinion is that we should be able to use the methodology developed for algebraic, axiomatic definition of abstract data types ([Guttag & Horning, 78]) as a high-level concept in a top-down development of the specifications of a system.

As an example, a text-editing system could be viewed as a data type 'text', which is defined by a set of functions that can create and manipulate the text. We should be able to give an axiomatic definition of these functions without deciding on any implementation details. Using this definition, we might discover the need for a subtype 'window' through which we can view and manipulate part of the text. This again might necessitate the definition of data types 'page', 'line' and 'character' and the functions that operate on these types. The axioms of the data types would be used to specify the predicates needed for correct program development/implementation of the functions.

Unfortunately, because of my lack of understanding of the concept of an abstract data type, and even more the lack of language constructs to express my ideas, this design philosophy was difficult to implement. It seems fairly easy to construct "stand-alone" data types, but it is more complicated to specify an integrated system of coordinated types. This note is primarily an attempt to clarify some of my basic thoughts on the subject. Therefore, the first part of it may be somewhat tutorial in form. Hopefully, the ideas presented in chapters 2 and 3 will be the basis of further, more fruitful work in the field.

1. DATA TYPES AND ABSTRACTION.

"Data types" and "abstract data types" are two of the basic concepts in this report. In theory, the only distinction between the concepts is some sense of hierarchy or levels of abstraction. In practical programming and systems development, however, this distinction may be useful and important, and I find it necessary to discuss it in some detail.

1.1. Abstraction.

According to [The Scribner-Bantam English Dictionary, 79], to abstract is

> "to derive or separate a general idea of, as a quality, from the object to which it belongs"

Using the same dictionary, we find that an idea is general only if it relates to a whole category or group of objects. Therefore, a straightforward definition of **an abstract data type** might say that it **defines a set of properties that are characteristic for a group of data types.** Unfortunately, this definition depends on an understanding of the concept of a data type and, since most data types are abstractions of types on a lower level, we have only succeeded in establishing a hierarchy of data types.

Example 1.1

> Integers in a certain range is a basic data type in most programming languages. This data type may be implemented on a machine using one's complement or two's complement binary representation. Integers as used in a programming language is clearly an abstraction of the two machine-level integer types.

1.2. Data Types.

A data type defines a set of data elements. However, it is more than a set. The following definition stresses that the elements must have some common, useful attributes, and it conforms with the explicit, functional specifications of data types introduced later in this report.

> A **data type** is a definition of the characteristic properties of, including the legal operations on, a set of elements that can be referenced and manipulated in a data process. An element belongs to a data type if it has the defined properties.

This definition covers "traditional" data types like 'integers' and 'characters', and it also allows types like 'files of records of a given type' and 'functions computing values of a given type'. Furthermore, it supports the view that "types are not sets" ([Morris, 73]), but rather a mechanism which among other things should enforce authentication (an element supplied to an operation should be consistent with the type expected by that operation) and security (any operation applied to an element should be meaningful for the type of element).

In the textbook on data types, [Horowitz & Sahni, 76] defines a data type as "the kind of data a variable may "hold" in a programming language". Even if this definition is too simple and vague for our purpose, it has one important ingredient: it defines a data type in the context of a programming language. First, this establishes a basic level of abstraction since the machine-level implementation and representation details of a data type are of no interest in a programming language. Secondly, it allows us to associate whatever property we want with a data type as long as we are able to express it with a programming language construct! Here it seems appropriate to quote [Rowe, 80]:

> "Language evolves when it is discovered that identical patterns of statements are being used many times in different programs. When this happens, a new language construct will be invented to describe the computation."

1.3. Abstract Data Types and Type Structures.

As outlined in the introduction, I would like to specify the modules of a system using data types that model the available information. To do this, I need a language where the following *abstract properties of a data type* can be explicitly defined:

(1.1) a) which elements belong to the data type
b) how the elements are created
c) how the elements relate to each other
d) which operations are allowed on the elements
e) the semantics of the operations
f) what exceptions there are to the application of operations on the elements

Most programming languages supply a set of basic data types like integer, real, boolean and character. For such types, all the properties in (1.1) are defined as part of the language. In some languages it is possible to extend the set of types using *type constructors*, like scalar, subrange, array, record, set and file. The set of elements of extended types (property (1.1.a)) are explicitly defined in type declarations, while the four other properties (1.1.b-f) are associated with the type constructor and are defined as part of the language.

Example 1.2

The PASCAL ([Jenssen & Wirth, 74]) declaration

array [1..10] **of** integer

defines a data type where the elements are arrays with 10 components of type integer. The general rules for creation, access and operation on the elements of this type are shared with all other arrays and are associated with the type constructor **array**.

A type constructor is used to construct a family of similar data types; it is an abstraction of the properties of these types. The information needed to construct a type is supplied through parameters. The name **abstract data type** usually denotes an explicit definition of a type constructor. Since a constructor defines the common structure of a set of similar types independent of the elements in the types, it would seem natural to use the name data structure as synonymous with abstract data type and type constructor. However, this name is used in a number of ways in informatics, and to avoid some confusion the name **type structure** or just **structure** will be used in the rest of the report.

Example 1.3

In the programming language ADA, the **package** construct can be used to define data types and structures. The following package defines the user interface of a queue structure ([Ichbiah & al., 79], page 13.9). User information is separated from implementation details through the use of interface packages and package bodies. This is an important abstraction feature, but the example illustrates that ADA is not a satisfactory language for the specification of abstract data types:

- The interface package contains implementation details (the QUEUE record). Only the person who implements the corresponding package body needs to know that the queue is to be stored in an array. Indeed, the decision to use such a storage scheme should be left to the implementer. It may be argued that the **private** clause protects the QUEUE record against all other users, and that the implementer is free to change the structure as long as the specified functions and procedures are supplied. However, this only strengthens the argument that there is no reason why a normal user should know the specific properties of the QUEUE record.

The reason for this anomaly is that a data structure in ADA is a definition of the storage scheme for an element of a type. The package construct does not define a type structure. It allows us only to connect a set of operations to a concrete, given storage scheme. Thus, ADA does not allow data abstractions where a structure is defined through functions which create and operate on the elements of a type.

- An interface package can only describe the operations on a data structure through a definition of more or less mnemonic names. The axioms and propositions needed for a correct implementation can be given as comments, but not as an integrated part of a structure definition.

```
(1.2) generic(type ITEM; SIZE: NATURAL)
      package ANY_QUEUE is
         restricted type QUEUE is private;

         function  EMPTY  (Q: in QUEUE) return BOOLEAN;
         procedure ADD    (Q: in ITEM; Q: in out QUEUE);
         procedure REMOVE (Q: in out QUEUE);
         function  FRONT  (Q: in QUEUE) return ITEM;

         QUEUE_OVERFLOW, QUEUE_UNDERFLOW: exception;

      private
         type QUEUE is
            record
               STORE:     array [1..SIZE] of ITEM;
               COUNT:     INTEGER range 0..SIZE := 0;
               IN_INDEX:  INTEGER range 1..SIZE := 1;
               OUT_INDEX: INTEGER range 1..SIZE := 1;
            end record;
      end;
```

Example 1.4

An algebraic, axiomatic definition of type structures ([Guttag & Horning, 78]) seems to be suitable for the purpose of this report. In the following example, a notation inspired by [Guttag, 77], [Rowe, 80], [Nakajima, Honda, & Nakahara, 80] and ADA is used. A description of the notation will be given in a later section. At this point, I will only remark on some properties of the example:

- The parameter 'max_size' and the functions 'full?' and 'size' are usually not included in an abstract definition of a queue. They are defined in this example to make the structure as similar as possible to the package ANY_QUEUE in example 1.3. Besides, it might be argued that a user should know the maximum size of a queue, even if this borders on implementation details.

- The error exceptions occur when an evaluation of 'size' after an operation would result in a value outside 'size_range'. If 'size' was evaluated in connection with all other operations on a queue, the errors would be discovered as violations of the subrange structure. Such extensive, implicit type-checking might be impractical. The **exceptions** clause allows us to list explicitly conditions that should be checked in connection with operations on the structure.

```
(1.3) type queue( item: type; max_size: natural ) =
        structure
          type size-range = 0..max-size;
          functions
           create( )                   returns queue;
           add    ( q:queue; x:item )  returns queue;
           remove( q:queue )           returns queue;
           front ( q:queue )           returns item;
           size  ( q:queue )           returns size_range;
           empty?( q:queue )           returns boolean;
           full? ( q:queue )           returns boolean;
          axioms
           A1: empty?( create( ) )   = true;
           A2: empty?( add( q; x ) ) = false;
           A3: remove( add( q; x ) ) =
                 if empty?( q )   -> create( );
                  □ ¬ empty?( q ) -> add( remove( q ); x );
                 fi;
           A4: front( add( q; x ) )  =
                 if empty?( q )   -> x;
                  □ ¬ empty?( q ) -> front( q );
                 fi;
           A5: size( create( ) )     = 0;
           A6: size( add( q; x ) )   = size( q ) + 1;
           A7: size( remove( q ) )   = size( q ) - 1;
           A8: full?( q )            = (size( q ) = max_size);
          exceptions
           E1: empty?( q ) ==> error( remove );
           E2: empty?( q ) ==> error( front  );
           E3: full?( q )  ==> error( add );
        end structure;
```

1.4. Consistency and Completeness of Type Structures.

By definition, axioms are statements of obvious truth. Therefore, to some extent, there is no way to defend an axiomatic specification of a data structure except by stating that the chosen set of functions and axioms describes the properties of the structure as I understand it. However, there are some objective rules that a specification should satisfy. The set of axioms should be **consistent** and **sufficiently complete.** These concepts are treated formally in [Guttag & Horning, 78]. The following introduction is based on the less formal descriptions in [Guttag, 80].

1.4.1. Consistency.

A set of axioms is consistent if it can lead to no contradictory[1] statement. In theory, it is an unsolvable problem to determine whether an arbitrary set of equations is consistent. However, it is often fairly simple to demonstrate consistency for a given set. For a data structure, this can be done by treating the equations of the specification as left-to-right rewrite rules and demonstrating that they exhibit the Church-Rosser property.

A set of rewrite rules has the Church-Rosser the result of applying one or more of the rules to reduce a term as far as possible is independent of the sequence in which the rules have been applied.

---

[1]A statement is contradictory if the equation **true** = **false** can be derived from it.

Example 1.5

The equations in the specification of a 'queue', (1.3), defines a 'language' in which

(1.4) front(add(add(remove(add(create();x1));x2);x3))

is a legal term. This term can be reduced by applying the axioms A1, A3 and A4 as rewrite rules:

```
  front(add(add(remove(add(create();x1));x2);x3))

= front(add(remove(add(create();x1));x2))              (using A4)

= if empty?  (remove(add(create();x1))) -> x2;          (using A4)
   □ ¬ empty?(          . . .          ) -> front(...);
  fi

= if empty?  (create()) -> x2;                          (using A3)
   □ ¬ empty?(create()) -> front(create());
  fi

= if true -> x2;                                        (using A1)
   □ false -> error(front);                             (using E1)
  fi

= x2
```

It is fairly simple to show that the axioms can be applied in the sequences <A3; A4; A4; A1> and <A4; A3; A4; A1> without affecting the final result.

1.4.2. Sufficient Completeness.

For a given abstract data type, the equations in the specification define a set of words or terms L(T). The terms are expressions that can occur in programs using the type. Obviously, the set of axioms must assign some meaning to such expressions, and the set is <u>sufficiently complete</u> if it assigns meaning to a certain subset of the expressions.

The set of functions in the specification of T can be partitioned into two subsets: S containing all functions whose range is T and O containing all functions whose range is other types.

Example 1.6

For the 'queue' in example 1.4, S = <create, add, remove> and O = <front, size, empty?, full?>

A given set of axioms, A, is a sufficiently complete axiomatization of T if and only if for any term $o(e_1;e_2;\ldots;e_n)$ in L(T) such that o belongs to O, there exists a theorem derivable from A with the form $o(e_1;e_2;\ldots;e_n) = u$ where u contains no functions of the type T.

Example 1.7

For the term defined in (1.4), example 1.5 shows that we can derive the theorem

```
front(add(add(remove(add(create();x1));x2);x3)) = x2
```

The axiom set in example 1.4 is sufficiently complete if similar theorems can be derived for all terms starting with one of the functions in <front, size, empty?, full?>.

2. ELEMENTS OF A SYSTEMS LANGUAGE.

The basic idea of this report is that the modules of a system can be specified by axiomatic definitions of abstract data types. However, I have found no language that allows the definition of such modules and the integration of the modules into a system. As a consequence, I find it necessary to suggest some new language constructs.

The constructs are inspired more by an intuitive feeling of what is needed in order to express the solution of a problem than a theoretical study of what properties a language should have. Since this report is intended as a basis for further, more theoretical, research, I make no excuse for this. My intention is to describe a language that allows

- axiomatic definitions of data types.
- checking of completeness and consistency of data types.
- definitions of transactions on data types and systems.
- integration of data types and transactions into a system.
- construction of application programs on a system.

The suggested language constructs are heavily influenced by PASCAL, ADA and Dijkstra's "mini-language" [Dijkstra, 76]. It is presumed that the reader is familiar with the standard features of these languages.

2.1. Systems, Transactions and Programs.

As a basis for the description of language constructs it is necessary to understand the concepts system, transaction and program.

- A **system** is a model of some available information. It contains a representation of the information (data) and it defines a set of transactions a user is allowed to perform in order to extract, transform or transmit some of the information.

A system is usually constructed as an integrated set of several data types, but a single data type like the 'queue' in example 1.4 might be considered a system by the definition given above.

- A **transaction** is a unit of processing in a system. It can extract, transform or transmit data. It is defined as a function or a procedure on an element of a data type and it is executed as one indivisible operation on a specific element of the type.

  Using the basic, axiomatically defined functions of a data type, we can construct new types of transactions. The definitions of such transactions will be separated from the definition of the data type (see section 2.5).

- A **program** is an ordered sequence of transactions on a system. A new transaction is defined by writing a program utilizing previously defined transactions.

In a standard programming language, a program is a high-level, independent unit containing the definitions of the data types, structures and the sequence of operations needed to solve a given problem. In a systems programming language, a program is a low-level unit defining a new transaction on an already defined system. The main intellectual effort is to develop a complete and consistent system with axioms and predicates defining the available transactions. Using the elements of a system, the construction of a new and correct program should be a less complicated process.

2.2. Data Types and Structures.

According to the definitions in sections 1.2-3, a *data type* defines the properties of a set of elements that can be manipulated in a data process, while a *type structure* is a definition of the abstract properties of a family of data types. This distinction defines a meaningful hierarchy between types and structures. However, it should be notedd that a structure may be considered as set of data types that can be referenced and manipulated to construct a system, and therefore a type structure has the properties of a data type whose elements are data types. The main feature distinguishing a type structure from other data types is that a structure will have a set of one or more parameters that are used to construct the elements of a new data type and that can be varied in order to define different data types.

New data types and structures can be defined in two ways:

a) By using previously defined structures to extend the set of data types and structures.

b) By giving an axiomatic definition of a new type structure.

Using a standard BNF-notation[1], the following syntactic form defines the two methods of type-definition.

```
(2.1) <type_definition> ::=
   a)         <type_declaration> |
   b)         <structure_specification>
```

---

[1] BNF stands for "Backus-Naur Form". A symbol enclosed in '<...>' denotes a syntactic class, while '[...]' encloses an optional expression. The symbol '|' is used to separate alternative syntactic forms.

2.2.1. Extending Data Types and Structures.

A <type_declaration> is defined by the following syntactic form:

```
(2.2) <type_declaration> ::=
        type <name> [ ( <list_of_parameters> ) ] =
          <structure_name> [ ( <list_of_arguments> ) ]
```

In (2.2) <structure_name> is the name of a previously defined type structure. Except for the use of parameters, this form of type definition is similar to the methods used in PASCAL and ADA.

Example 2.1

The declaration

```
type int_array_10 = array( 1..10; integer );
```

defines a data type whose elements are arrays with 10 components containing integers. 'array' is the name of a structure with two parameters: 'index_range' and 'value_type'. In the declaration the parameters are given the argument-values '1..10' and 'integer'.

Similarly, the declaration

```
type array_10( value_type:type ) =
        array( 1..10; value_type );
```

defines a new type structure with one parameter: 'value_type". The elements of the structure are data types containing arrays with 10 components.

An extensive system of parameterization of type structures is one of the basic features of the language. It may be difficult to implement, but it is needed to express the interaction between types. At this point, the descriptive qualities of the language are more important than the possibility of an effective implementation. Therefore, implementation problems will be ignored.

The notation differs somewhat from standard PASCAL or ADA. In order to have a common notation for all structures, the parameters are enclosed in parentheses. The use of the Roman font instead of the bold font for the structure_name ('array' instead of '**array**') illustrates that all structures can be explicitly defined instead of being special, pre-defined features of the language.

2.2.2. Axiomatic Definition of Type Structures.

The syntactic class <structure_specification> (2.1.b), has the following general form:

```
(2.3) <structure_specification> ::=
        type <name> ( <list_of_parameters> ) =
          structure
           [ use <list_of_external_data_types>; ]
           [ <list_of_type_declarations>; ]
            functions
              <list_of_functions>;
            axioms
              <list_of_axioms_defining_the_functions>;
           [ exceptions
              <list_of_exceptions_to_the_application_of_functions>; ]
          end structure
```

The definition of a 'queue' in example 1.4 demonstrates the use of this syntactic form.

Some comments:

- A 'list' is a sequence of elements separated by ';'.
- The '**use**'-construct is generally used to introduce elements (data types, variables, etc.) that are defined external to a module. The elements cannot depend on the parameters of the module.
- The <list_of_functions> should contain primarily <u>cardinal</u> functions, i.e. functions that are needed to define the structure and

that can be used to define all other transactions on the structure. This requirement will be discussed in later sections.

2.3. Some Basic Type Structures.

As an illustration of the use of the language and as a basis for the example in chapter 3, the type structures identifier, pointer, record and file will be defined explicitly. I will presume that some basic data types like integer, boolean and character are available, and the PASCAL subrange structure will be used without definition. To be consistent with the syntactic form (2.1.a), an integer subrange like '1..10' should be defined by an expression like 'subrange(integer; 1; 10)'. However, the former expression will be allowed as a shorthand notation. Such well established and unambigious shorthand notations will also be introduced for other functions and structures.

2.3.1. Identifiers.

An **identifier** is defined as a string of characters that **can be used as a name for an element of a data type.** According to section 1.2, any set of elements that can be referenced and/or manipulated in a data process is a data type. This means that a set of programs/procedures, a set of statements, a set of files and a set of integers are examples of valid data types and that we can define and distinguish between different types of identifiers like program-names, statement-labels, file-names and integer variables. Clearly, such sets of identifiers satisfy the condition to be data types. Furthermore, all sets of identifiers have some common properties:

- An arbitrary element of a type can be attached to a given identifier of the type.
- An identifier is defined if and only it a valid element is attached to it.
- An attached element can be referenced through the identifier.

The conclusion of this argument is that we can define a type structure 'identifier', which specifies these abstract properties and which can be used to construct sets of identifiers of any type.

```
(2.4) type identifier( datatype:type ) =
        structure
          functions
            create  ( datatype )     returns identifier;
            assign  ( A:identifier; x:datatype )
                      returns identifier;  --alias A := x ;
            use     ( A:identifier )
                      returns datatype;    --alias A ;
            defined?( A:identifier ) returns boolean;
          axioms
            A1: defined?( create( datatype ) ) = false;
            A2: defined?( assign( A; x ) )     = true;
            A3: use     ( assign( A; x ) )     = x;
          exceptions
            E1: ¬ defined?( A ) ==> error( use );
        end structure;
```

Some comments:

- It should be noted that I consider 'identifier' to be an abstract concept that includes the concept of a 'variable'. This may be somewhat controversial because an identifier is often defined as just a string of characters with no inherent meaning, while a variable is a combination of an identifier and a value. I find it difficult to accept this distinction. An identifier must quite naturally identify some object. That is, it must be understood and used as a reference to the object. This is also how I understand the concept of a variable, but whereas an identifier can be a static reference to some constant element like a program, a variable is dynamic in the sense that it can refer to different elements during a data process.

- The 'assign'-function is usually associated with variables. However, the semantics of the function is the same whether we use it to attach a name to a type in a type-declaration or an integer value to a variable during the execution of a process. Since the 'identifier'-structure will be used mostly to define variables, I have decided to use this well known name for the function.

- Wherever it seems appropriate, I will allow accepted shorthand notations instead of the functional notations defined in the structure-specification. The **alias**-construct, as used in (2.4), is meant to imply that wherever an expression with the form 'A := x' occurs, it should be interpreted as 'assign(A; x)'. Likewise, I will allow the use of <identifier> as an alias for 'use(<identifier>)'.

- The following syntactic form is commonly used to declare <string_of_characters> as an identifier of a type:

(2.5) <string_of_characters> : <name_of_type>

In *my* language, however, this defines <string_of_characters> as an element of the type, while an identifier should be defined by

(2.6) <name_of_identifier> : identifier(<name_of_type>)

This important distinction must be observed in all <structure_specifications>. In programs and packages (see section 2.5) however, it may not be necessary to distinguish between an element and an identifier of a type and when it can lead to no misunderstanding, I will allow the use of (2.5) as an alias for (2.6).

- The only legal use of an element of a type is as a parameter to the functions defined for the type. Thus, an identifier can only appear in the functions 'assign' and 'use' or in one of the aliases for these functions. It is very important to understand that an <identifier> appearing in any other context is an alias for 'use(<identifier>)' and that all operations involving such aliases are operations on the elements produced by the 'use'-function. For example, the expression 'assign(A; B)', where A and B are identifiers, must be interpreted as 'assign(A; use(B))'. Without this requirement, we would have to repeat the definition of all functions of a type for identifiers of that type. This would make it impossible to specify a common, abstract 'identifier'-structure.

- All data structures will be defined with a 'create'-function. This function is invoked to create an element of a data type and will attach all the properties of the structure to the element. The abstract properties of a structure may be independent of any of the parameters to the structure. Therefore, the 'create'-function is often defined without parameters. I have included 'datatype' as a parameter to the 'create'-function above to allow different implementations of identifiers, depending on the 'datatype'.

As an end to this section, I will comment on the concept of variables and assignment. [Icbiah & al., 79] classifies variables as static or dynamic. A static variable is usually implemented as a reference to a memory location where an element of a given type can be stored. During the execution of a process, different elements can be moved or "assigned" to the location. In such implementations, the reference is static and the elements are the varying or dynamic part of an assign-operation.

For dynamic variables, the reference value varies, while the elements may remain in fixed locations throughout the execution. An element is assigned to a variable by changing the reference-value to "point" to the location of the element. The use of pointer variables, as defined in the next section, allows dynamic references to elements, but the structure may be unecessesary complicated because it involves two levels of referencing, which are both accessible to the user: a pointer is a dynamic reference to a static reference to an element.

The semantics of (2.4) is valid for static variables and for dynamic variables, where the second level of referencing is hidden from the user. Furthermore, I can think of no data type where the category of the variables is part of the abstract properties of the type. I suspect that the choice between static and dynamic variables for a type can be left to the implementer, and that the distinction between pointers and ordinary variables is unnecessary on the specification and application programming level of systems construction. However, the next section of the report is retained to demonstrate that the 'pointer'-structure can be easily specified in my language.

2.3.2. Pointers.

As described in the previous section, a variable is a special type of identifier. In many situations it may be easier to use this more familiar name, and to allow this we can define 'variable' as a type structure that is identical to 'identifier':

```
(2.7)  type variable( datatype: type ) = identifier( datatype );
```

Since the set of variables of a type constitutes a separate data type, it is quite natural to allow identifiers of this variable-type. Such composite identifiers point to an element of the basic data type through a variable of the type. Using the syntactic form (2.2), the structure of pointers can be defined by:

```
(2.8)  type pointer( datatype: type )
                = identifier( variable ( datatype ) );
```

The two structure-names 'identifier' and 'variable' are used to distinguish between the two levels of referencing.

Substituting 'pointer' for 'identifier' and 'variable' for 'datatype' in (2.4), we get the following set of 'pointer'-functions:

```
(2.9)  create  ( variable )              returns pointer;
       assign  ( P: pointer; A: variable) returns pointer;
       use     ( P: pointer )            returns variable;
       defined?( P: pointer )            returns boolean;
```

Qualifying these function-names by 'pointer' and the original functions of (2.4) by 'variable', we can write the following legal expressions:

```
(2.10) a) pointer.assign( P; variable.create( datatype ) );
       b) pointer.assign( P; variable.assign( A; x ) );
            -- x is an element of type 'datatype'
       c) variable.assign( pointer.use( P ); x );
       d) variable.use( pointer.use ( P ) );
       e) pointer.defined?( pointer.create( variable ) );
       f) pointer.defined?( pointer.assign( P; A ) );
```

These expressions reflects the normal use of pointers:

a) Expression (2.10.a) corresponds to the function 'new' in PASCAL. It creates an undefined variable of the basic datatype and assigns a reference to this variable to the pointer P.

b) Expression (2.10.b) creates a pointer-reference to an element x of the basic data type. This operation is not defined in PASCAL or ADA, but it corresponds to the 'β'-function defined in [Wulf & al., 81, page 208].

c) Expression (2.10.c) corresponds to the statement 'P↑ := x' where 'P↑' is an alias for 'pointer.use(P)'.

d) Expression (2.10.d) corresponds to the term 'P↑' used on the right-hand side of the assignment operator in PASCAL. 'P↑' is still an alias for 'pointer.use(P)', but the result of this function is a variable and by (2.4), a variable-name can be used as an alias for the function 'variable.use'. This function returns an element of the basic data type and therefore, 'P↑' used on the righthand side of assignment must be interpreted as a reference to this element.

e) Using the axioms of (2.4), the value of (2.10.e) should be **false** while the value of (2.10.f) should be **true**. However, this is in conflict with the concept of an undefined pointer in PASCAL and reflects the one point where (2.8) and (2.9) give an unsufficient specification of the 'pointer'-structure. It should be possible to distinguish between three situations:

   1) The pointer contains a reference to no element of the basic data type. This is the case after a pointer has been created, but before it has been assigned a value. In PASCAL, it is also the case after a pointer has been assigned the reference-value **nil**. A pointer with this value is considered to be defined, but it is empty or a <u>null-pointer</u>.

   2) The pointer contains a reference to a variable that is undefined. That is, the variable is created, but has been assigned no element of the basic type. This is the case after the execution of the function 'new' or the expression (2.10.a). Such pointers are called <u>undefined</u>, but they are not null-pointers.

   3) The pointer contains a reference to a variable that is defined. Such pointers are defined and they are not null-pointers.

   To give an axiomatic definition of these cases, we must introduce the function 'null?' and the value **nil** and change the semantics of 'defined?' according to the following axioms:

```
A1: null?    ( assign( P; nil ) )  = true;
A2: use      ( create( variable ) ) = nil;
A3: defined?( new ( P ) )          = false;
A4: defined?( assign( P; A ) )     = variable.defined?( A );
```

It is fairly easy to write an explicit specification that defines all the abstract properties of pointers. However, the points above should illustrate that the definition of aliases and functions to conform with the PASCAL-notations might become somewhat complicated. Since I am not convinced that the 'pointer'-structure is useful at the specification level of systems development, I will refrain from introducing this extra complication.

2.3.3. Records.

The 'record' is a well-established and presumably well-understood programming language construct. Even so, it may be one of the more complex structures to specify formally. Some of the properties of a record are:

- A record is a collection of components. A component identifies an element of a data type, and each component can be of a different type. The components can only be accessed through the record.
- The components of a record are ordered in a sequence. Operations on a single component involve some form of indexing of the components.
- The number of components in a record can vary and should be a parameter to the specification of a general record-structure.

The key to understanding the following specification is that a component is defined as an identifier of a type. Indeed, the definition of 'identifier' as a structure was inspired by the difficulties encountered in giving a formal specification of the 'record'-structure. The problem was that both the name and the type of a component were needed as parameters to a record and there was no formal description of a name and its connection to an element of a data type.

Thus, a record is defined as a collection of identifiers (variables) of elements rather than a collection of elements. It should be remembered that the structure specifies the abstract properties of a record and not the implementation. These properties include the ability to reference a record as a unit, construct a record from given elements of the component types and reference the component elements through the record and component names.

The problem of indexing and varying the number of components is not properly solved. Basically a record is defined with a fixed number, n, of components. This can be used as a model for the definition of records with k components where k can be any positive integer.

```
(2.11) type record(component_1:identifier(datatype_1:type);
                   component_2:identifier(datatype_2:type);
                   . . .
                   component_n:identifier(datatype_n:type)) =
       structure
         functions
           create    ( component_1; ... component_n)
                                           returns record;
           construct(R:record; x_1:datatype_1; ... x_n:datatype_n)
                                           returns record;
           defined? (R:record)             returns boolean;
           assign    (R:record; component_i; x_i:datatype_i)
                                           returns record;
                 --alias R.component_i:=x_i;
           use       (R:record; component_i) returns datatype_i;
                 --alias R.component_i;
           equal?    (R1:record; R2:record) returns boolean;
                 --alias R1 = R2;
        axioms
           A1: defined?(create(component_1; ... component_n)) = false;
           A2: defined?(construct(R;x_1; ... x_n))            = true;
           A3: construct(R;x_1; ...x_n) =
                 assign(assign(...assign(create(component_1;...component_n);
                        component_1;x_1)...); component_n;x_n);
           A4: use( assign( R; component_i; x_i); component_j) =
                 if   i = j -> x_i;
                 □    i ≠ j -> use( R; component_j);
                 fi;
           A5: equal?( create( component_1; ... component_n );
                       create( component_1; ... component_n ) ) = true;
           A6: equal?( assign( R1; component_i; x_i )
                       assign( R2; component_i; y_i ) ) =
                 ( x_i = y_i )
               ∧ ( ∀k, 1≤k≤n, k≠i | R1.component_k = R2.component_k )
         exceptions
           E1: ¬ defined?( R ) ==> error( use );
       end structure;
```

2.3.4. Files.

An axiomatic definition of a data type must be based on a model of the elements of the type. For the 'file'-structure, my model has the following properties:

- A file is a sequence of zero or more components.
- A component is an element of some data type.
- Only the first component in a non-empty file is accessible to a user.
- The sequence following the first component of a non-empty file is a file.
- A new component can be added to the end of a file.

A component of a file is usually called a record, but I have used the name 'component' to stress that it is not necessarily a record as defined in (2.11).

```
(2.12) type file( component:type ) =
         structure
            functions
              create    ( )                        returns file;
              add       ( f:file; x:component )    returns file;
              front     ( f:file )                 returns component;
              tail      ( f:file )                 returns file;
              empty?    ( f:file )                 returns boolean;
              equal?    ( f:file; t:file)          returns boolean;
              member_of?( f:file; x:component)     returns boolean;
              part_of?  ( f:file; t:file )         returns boolean;
            axioms
              A1 : empty?( create( ) )    = true;
              A2 : empty?( add( f; x )    = false;
              A3 : front ( add( f; x ) ) =
                               if   empty?( f ) -> x;
                               □ ¬ empty?( f ) -> front( f );
                               fi;
              A4 : tail  ( add( f; x ) ) =
                               if   empty?( f ) -> create( );
                               □ ¬ empty?( f ) -> add( tail( f ); x );
                               fi;
              A5 : equal?( create(); create() )        = true;
              A6 : equal?( create(); add( f; x ) )     = false;
              A7 : equal?( add( f; x ); add( t; x' ) ) =
                                  ( x = x' ) ∧ equal?( f; t )
              A8 : equal?( f; t )   ==> equal?( tail( f ); tail( t ) );
              A9 : part_of?( f; t ) =
                                  equal?( f; t ) ∨ part_of?( tail( f ); t );
              A10: member_of?( f; front( t ) ) = part_of?( f; t );
            exceptions
              E1: empty?( f ) ==> error( front );
        end structure ;
```

Some comments:

- The functions 'part_of?' and 'member_of?' are included because they are needed to specify the predicates for some transactions on files.

- The four axioms for 'equal?' illustrate that a complete set of axioms must define the semantics of a function for all possible parameters to the function. The parameters to 'equal?' are 'files', which can be the result of a 'create', 'add' or 'tail'-function. The axioms defines 'equal?' for all possible 'files'.

2.4. Cardinal Functions.

As mentioned in section 2.2.2, the set of functions defined in a structure specification should be cardinal. The objective properties of this concept should be studied further. Here, I can only offer an intuitive and informal description, motivated by the fact that the concept has been useful in the development of a system.

- A **cardinal set of functions** defines all the properties we associate with a data structure. All operations or transactions on elements of a data type with a given structure can be defined using one or more of the cardinal functions.

Example 2.2

It is well known that the set of functions {and, or, not} is cardinal for the boolean data type.

Some of the important features of the concept are:

- The structure-specification need only define a cardinal set of functions. The construction of other operations may be separated from the specification and delayed to allow a stepwise refinement of a module of a system.

- All multi-range functions (i.e. functions which yield more than one result value) can be constructed using one-range functions. Thus, the specification of a structure can be simplified if only one-range cardinal functions are allowed.

- A structure defines the abstract properties of a set of data types. Operations of interest for one specific use of a data type can be separated from the structure-specification.

2.5. Packages of Transactions.

As a consequence of the arguments in section 2.4, we need a language construct that can be used to define a set of transactions and attach them to an element of a given type. This construct will be called a 'package'. (2.13) is a definition of the syntactical form of a package. The details of the form will be shown in an example where a set of transactions on a file will be defined.

```
(2.13) <package_specification> ::=
          package <package_name> =
            attach to <element>:<type>;
          [ use <list_of_external_types>; ]
          [ <declarations_of_local_types_and_identifiers>; ]
            transactions <list_of_transactions>;
          end package
```

Some comments:

- A package is attached to an element of a given type by a declaration-statement of the form:

  <element_name>:identifier(<actual_type>) **with** <package_name>;

  The <element> and <type> in the **attach**-statement are the only parameters to a package. <type> is usually a reference to a structure-specification, and <actual_type> must be of this given structure. The <element_name> will replace <element> in the package and is the only legal argument to a transaction.

- The **use**-statement is included to make it possible to import externally defined types.

- A package will usually include a set of local identifiers/variables that can be manipulated by all the transactions. All elements declared with the package will have their own copies of these variables.

- A transaction is a program constructed by using the set of cardinal functions of the given <type>. It can be either a procedure or a function. A procedure can operate on the element to which the package is attached and on the local identifiers, but it cannot return any result to the calling program. The only way to return a result to a calling program, is through a function-transaction.

2.5.1. Transactions on a File.

As an example of a package, some transactions on a file will be defined. The functions in (2.12) are sufficient to construct a file and to partition it into a 'front'-component and a 'tail'-file. However, they offer no way of reading the successive components of a file. To do this requires keeping track of a 'current' component that is available to a data process and the 'rest' of the file following this component. The 'read'-transaction can operate on these variables. The package will also contain a transaction that 'resets' the file so that the first component is the next to be 'read', a transaction that returns the 'current' component to the data process, and a transaction 'eof?' to test whether the end of the file has been reached.

```
(2.14) package file_transactions =
        attach to f:file(component:type);
        type      file = file(component:type);
        ident      current: component := front(f);
                   rest   : file      := f;
                   end    : boolean   := false;
        transactions
          procedure reset;
                begin
                  { true }
                    rest := f;
                  { front( rest ) = front( f ) }
                 end;
          procedure read;
                 begin
                  { part_of?( f; rest ) }
                    if  empty?( rest ) ->  end := true;
                                          current := component.create( );
                     □ ¬ empty?( rest ) -> current := front( rest );
                                          rest := tail( rest );
                    fi;
                  { end = empty?( rest' ) ∧ ¬ end ==>
                    ( member_of? ( f; current ) ∧ part_of?( f; rest )  ∧
                      current = front( rest' ) ∧ rest = tail( rest' )) }
                  end;
          function record returns component;
                  begin
                   { member_of?( f; current ) }
                     record := current;
                   { member_of?( f; record( f ) ) ∧ record( f ) = current }
                  end;
          function eof? returns boolean;
                  begin
                   { true }
                     eof? := end;
                   { eof?( f ) = end }
                  end;
      end package;
```

Some comments:

- The only parameter to a transaction is the name (identifier) of the element to which it is attached. Since this parameter is implied by the inclusion of a transaction in a package, it is not necessary to specify it explicitly for each transaction.

- The keyword **ident** precedes the identifier-declarations and may be interpreted as an alias for explicit references to the 'identifier'-structure.
- The explicit testing of the condition 'empty?(f)' in 'read' is included to facilitate the implementation of 'eof?'. This transaction should yield the value **true** only after an attempt to read past the end of the file. If 'eof?' is not needed, it might be sufficient to let the function 'front' raise the error-exception E1 of (2.12). Somewhat arbitrarily, the undefined element 'create()' is assigned to 'current' if 'rest' is empty.
- Transactions are programs and should be presented with predicates defining their semantics. It seems reasonable to demand that all transactions in a package can be executed under any conditions as long as the package is attached to an element of the correct type. This implies that the preconditions of the transactions should be **true** after a successful initialization. Example 2.3 shows that this is the case for the 'read'-transaction in (2.14).

Example 2.3

The 'read'-transaction operates on the 'rest' of the file, i.e. the part of the file not yet read. Informally, we can state the following postconditions for the transactions:

a) If the 'rest' of the file is empty, the transaction should result in an end_of_file-condition.

b) If the 'rest' of the file is not empty, the transaction should result in a 'current' component which is a member of the original file and equal to the front-component of the 'rest' of the file. The 'rest'-file should be updated to be a part of the original file and the tail of the former 'rest' of the file.

The postcondition can be formally stated as:

```
(2.15) a)   end = empty?( rest' ) ∧ ¬ end ==>
       b)   (  member_of?( f; current )
             ∧ part_of?( f; rest )
             ∧ current = front( rest' )
             ∧ rest    = tail( rest' ) )
```

The file 'rest' marked with an apostrophe, refers to the value before the transaction is executed.

(2.15.a) follows from the semantic properties of the **if**-construct. The weakest precondition for (2.15.b) can be found using predicate transformers (Dijkstra, 76) and the axioms of (2.12):

```
wp('read',(2.15.b))

  =   member_of?( f; front( rest' ) )
    ∧ part_of?( f; tail( rest' ) )
    ∧ front( rest' ) = front( rest' )
    ∧ tail( rest' )  = tail( rest' )

  =   member_of?( f; front( rest' ) )
    ∧ part_of?( f; tail( rest' ) )

  =   part_of?( f; rest' )              ( by axiom A10)
    ∧ part_of?( f; tail( rest' ) )

  =  part_of?( f; rest' )               ( by axioms A8, A9 )
```

Hence, the precondition is an invariant of the transaction, and will always be true if it is true before the first execution. In the declaration of 'rest', the file is assigned the value 'f' before the first execution. Since 'part_of?(f; f) = **true**' (axiom A8), the precondition will always be true if the package is attached to a valid file.

2.6. Systems.

A <system_specification> should contain a definition of all type structures, data types and identifiers that are used to represent the information in the system. The following syntactical form allows such definitions:

```
(2.16) <system_specification> ::=
          system <name> =
            [ use <list_of_external_types_and_packages>; ]
              type <list_of_type_definitions>;
            [ package <list_of_package_specifications>; ]
              ident <list_of_identifier_declarations>;
          end system;
```

The example in chapter 3 will show how the modules of a system can be developed and incorporated. A <system_specification> defines all legal transactions on the system but, except for the initializations associated with the declarations of identifiers, it specifies no processes to be executed. Such processes must be defined in separate programs.

3. CONSTRUCTING A SYSTEM. AN EXAMPLE.

A system will usually incorporate a large number of different data types with complex interrelations into a unified model of some available information. In most cases, the analysis, development and maintenance of the information-model is separated from the development of programs that extract and manipulate systems-data to solve a given problem. The approach in this note is to develop a system with a basic set of transactions. The transactions are the only legal way to access the data in the system, and application programs must be written in terms of the transactions.

As usual, it is difficult to find examples which are complex enough to exhibit the properties of the theory, and small enough to present in a short paper. The "Welfare Crook Problem" seems to be a good compromise between these requirements. The problem was presented to me by David Gries. It is fairly small, easy to understand, and has a short and elegant solution by "traditional" programming metdods. Using the systems approach to find a solution may not be worth the effort, but it illustrates the interactions and incorporation of a number of different data types, and the use of functions, axioms and transactions to develop correct programs.

3.1. The Welfare Crook Problem.

The following three files are given:

- One file contains information on all employees of a corporation.
- One file contains information on all students receiving financial aid from a university.

- One file contains information on all unemployed persons in a certain area who receive welfare-benefits.

It is suspected that one or more persons are registered in all three files. Develop a system that permits an investigation of the set of files to find such "welfare crooks".

3.2. Basic Data Types.

In this section the available information is analyzed and structured without reference to the problem to be solved. The definition of the problem describes three categories of persons. A person can be represented by a record containing an identification and some information which depends on the organisation the person is connected to. The identification may also vary from organisation to organisation, but since we must be able to compare identifications from different organisations, it will be presumed that they are all of the same type. On this basis, we can define the following structure for the personnel-records:

```
(3.1) type person( information: type ) =
            record( id: identification; info: information);
```

Using this structure, the following three record types will represent the personnel registered in the three files.

```
(3.2)
   a) type CORP_employee = person( CORP_information );
   b) type UNIV_student  = person( UNIV_information );
   c) type WEL_recipient = person( WEL_information );
```

The three files will have the following structure:

```
(3.3) type personnel_file( information: type ) =
            file( person( information: type ))
                                        with file_transactions;
```

where 'file_transactions' is the package (2.14) of transactions needed to operate on the files. The files can be declared by

```
(3.4)
   a)       CORP: personnel_file( CORP_information );
   b)       UNIV: personnel_file( UNIV_information );
   c)       WEL:  personnel_file( WEL_information );
```

3.3. Data Types.

The system is constructed to allow an investigation of a set of three files. It seems natural to define a data type 'set_of_files' and describe the investigation in terms of transactions on an element of this type. The type should have the following properties:

- A 'set_of_files' can be referenced as a unit.
- A 'set_of_files' can be constructed from any three files.
- Any of the component-files can be selected and operated on with defined file-transactions.

Some thought will show that a 'set_of_files' will have all the abstract properties of a 'record' as defined in (2.11), and there are no reasons to avoid using this structure to specify 'set_of_files':

```
(3.5) type set_of_files( f_1: identifier( file_1:type );
                         f_2: identifier( file_2:type );
                         f_3: identifier( file_3:type ) )
                       = record( f_1; f_2; f_3 );
```

After defining the structure 'set_of_files' we can construct transactions on elements of the type. At this point, we may not know all the transactions needed to solve the given problem, but knowing that the components are files, it seems reasonable to define transactions for the following operations:

- Basically, only the 'front'-record of a file is available to a user. Attaching the package (2.14) of 'file_transactions' to a file, we can read the consecutive 'current' records and make them available to a user through the transaction 'record'. For a 'set_of_files' there should be a transaction to <u>extract</u> the 'current' record from each of the component files and make the set of three records available to the user.
- The 'current' record of a component file is undefined after an attempt to read past the end of the file. There should be a transaction to test whether a 'set_of_files' is <u>complete</u> in the sense that no such attempt has been made on any of the component files.
- Files can be reset to read the first record. There should be a transaction to <u>reset</u> a 'set_of_files' to extract the set of first records from the component files.

It is clear that all the files in our system must be defined with the package (2.14) of 'file_transactions'. Furthermore, the result of the 'extract'-transaction is a data type 'set_of_records' which is still undefined. This is an important data type, because an element of this type will contain the only records of the system to which a user will have access. Like 'set_of_files', it comforms to the 'record'-structure of (2.11).

```
(3.6) type set_of_records( r_1:identifier( record-1:type );
                           r_2:identifier( record_2:type );
                           r_3:identifier( record_3:type ) )
                         = record( r_1; r_2; r_3 );
```

Now we are ready to define a package of transactions on a 'set_of_files':

```
(3.7) package set_transactions =
      attach to sof: set_of_files
                ( f_1: file( record_1:type ) with file_transactions;
                  f_2: file( record_2:type ) with file_transactions;
                  f_3: file( record_3:type ) with file_transactions );
      type records = set_of_records( r_1:record_1; r_2:record_2;
                                                        r_3:record_3 );
      ident current: records;
      transactions
        procedure reset;
          begin
           { true }
           file.reset( sof.f_1 );
           file.reset( sof.f_2 );
           file.reset( sof.f_3 );
           { ∀i, 1≤i≤3 | file.record( sof.f_i ) = file.front( sof.f_i ) }
          end;
        function complete? returns boolean;
          begin
           { true }
           complete? := ¬ ( file.eof?( sof.f_1 )
                          ∨ file.eof?( sof.f_2 )
                          ∨ file.eof?( sof.f_3 ) );
           { complete?( sof ) = ( ∀i, 1≤i≤3 | ¬ file.eof?( sof.f_i ) }
          end;
        function extract returns records;
          begin
           { true }
           if complete?( sof ) ->
              extract := records.construct( current; file.record( sof.f_1 );
                             file.record( sof.f_2 ); file.record( sof.f_2 ) );
            □ ¬ complete?( sof ) ->
              extract := records.create( r_1; r_2; r_3 );
           fi
           { complete?( sof ) ==> extract( sof ) = records.construct( current;
                         file.record( sof.f_1 ); file.record( sof.f_2 );
                                                   file. record( sof.f_3 ) );
          end;
     end package;
```

Note that while there are no constraints on the type of files in a general 'set_of_files', the package 'set_transactions' can only be attached to a set where the component files have some specific properties.

3.4. The System.

As a summary of the previous sections, the following system can be specified. The specific properties of the types imported by the 'use'-construct are of no interest at this point, and they may be considered as defined externally. To avoid repetitions, the packages 'file_transactions' and 'set_transactions' are also considered to be defined externally. Standard structures like 'record' and 'file' are considered to be globally defined.

```
(3.8) system file_investigation =
        use type
          identification;
          CORP_information; UNIV_information; WEL_information;
        use package
          file_transactions; set_transactions;
        type
          person( information:type ) =
            record( id:identification; info:information );
          personell_file( information:type ) =
            file( person( information:type ) with file_transactions;
          set_of_files( f_1: identifier( file_1:type );
            f_2: identifier( file_2:type );
            f_3: identifier( file_3:type ) )   = record( f_1; f_2; f_3 );
          set_of_records( p_1: identifier( record_1:type );
            p_2: identifier( record_2:type );
            p_3: identifier( record_3:type ) ) = record( p_1; p_2; p_3 );
        ident
          SYSTEM: set_of_files( CORP: personell_file( CORP_information );
                                UNIV: personell_file( UNIV_information );
                                WEL:  personell_file( WEL_information ) )
                                with set_transactions;
          PERSONS: set_of_records( CORP_employee: person( CORP_information );
                                   UNIV_student:  person( UNIV_information );
                                   WEL_recipient: person( WEL_information ) )
                                   := extract( SYSTEM );
      end system;
```

3.5. Developing a Program.

A system defines an environment of data types, identifiers and transactions that can be used to construct a program that manipulates the data in the system. The goal is to express all operations and predicates of a program in terms of the transactions and functions defined for the data types in the system. To attain this goal, we may have to define some new transactions and expand the set of packages attached to the data types. We can do this without changing the basic structure of the types and this shows that the language is well suited for a stepwise refinement of system and programs.

As an illustration of this, we can develop a program to find all "Welfare crooks" in the system (3.8). A first version of the program may be stated as:

```
(3.9) program find_crooks;
        use system file_investigation;
        begin
            { Given a SYSTEM of three 'personnel_files'
              as defined in 'file_ivestigation' }

              Find all crooks in the SYSTEM;

            { All the persons in the SYSTEM have been investigated
              and all the crooks have been identified }
        end;
      end program;
```

Since a crook is a person registered in all three files of the SYSTEM and we can extract only one set of PERSONS at any time, the basic operation to find a crook must be to test whether all the records in PERSONS represent the same person. This test can be defined as a transaction 'same_person?' on an element of the type 'set_of_records'. To shorten the presentation, the transactions 'less_1?', 'less_2?' and 'less_3?'

are included in the following package. The need for these transactions will appear later in the analysis of the program.

```
(3.10) package person_transactions =
        attach to
         sop: set_of_records( p_1: person( info_1:type );
                              p_2: person( info_2:type );
                              p_3: person( info_3:type ) );
        ident maxid: identification := max( p_1.id; p_2.id; p_3.id );
        transactions
         function same_person? returns boolean;
          begin
           { defined?( sop ) }  --See (2.11) for definition of 'defined?';
           same_person? := p_1.id = p_2.id ∧ p_1.id = p_3.id;
           { same_person( sop ) = ( p_1.id = p_2.id ∧ p_1.id = p_3.id ) }
          end;
         function less_1? returns boolean;
          begin
           { maxid = max( p_1.id; p_2.id; p_3.id )}
           less_1? := p_1.id < maxid;
           maxid := max( p_1.id; p_2.id; p_3.id );
           { less_1?( sop ) = (p_1.id < maxid')
             ∧ maxid = max( p_1.id; p_2.id; p_3.id ) }
          end;
         function less_2? returns boolean;
            ... --The function is similar to 'less_1?'
         function less_3? returns boolean;
            ...
       end package;
```

Some comments:

- The transactions above can only operate on a 'set_of_records' where the records has an 'id'-component. This illustrates the need to separate the transactions for a special application from the cardinal functions that define the general structure of a data type.

To find all crooks in the SYSTEM, the test 'same_person?' must be repeated as long as there are more sets of PERSONS left to investigate. This can be stated as a **do**-loop with the informal precondition { "more persons to investigate" }. The precondition {Given a SYSTEM ...} can be formalized to { defined?(SYSTEM) }. Using the definition in (2.11), this means that all three components of the SYSTEM have been assigned a valid element of the type 'personnel_file'.

After this analysis, we can write a new version of the program 'find-crooks', where the preconditions for the operations (as defined in (3.7) and (3.10)) are included. Operations that are still undefined are enclosed in quotes.

```
(3.11) program find_crooks;
        use system file_investigation;
        use type CROOK_information;
        ident CROOK: person( CROOK_information );
        begin
         { defined?( SYSTEMS ) }
         do { "more persons to investigate?" }
            { true }            -- precondition for 'extract';
            PERSONS := extract( SYSTEM );
            { defined?( PERSONS ) }   -- precondition for 'same_person?';
            if same_person?( PERSONS ) ->
                        "identify CROOK";
                        "move SYSTEM to extract a new set of PERSONS";
             □ ¬ same_person?( PERSONS ) ->
                        "move SYSTEM to extract a new set of PERSONS";
            fi;
         od;
         { All persons in the SYSTEM are investigated
           and the CROOKs have been identified. }
        end;
       end program;
```

Using the definition of 'extract', it is easily shown that

wp("extract(SYSTEM)", defined?(extract(SYSTEM))) = complete?(SYSTEM)

Hence, {complete?(SYSTEM)} is a preconditon for the body of the loop in (3.11).

At this point, we are ready to discover (surprisingly?!) that the further refinement of the program will be much easier if the files in the SYSTEM are ordered. That is, the persons are registered in ascending order of the identification-field 'id'. Without further argument, I will assert that this leads to the following precondition for the **do**-loop:

(3.12) The SYSTEM consists of ordered personnel_files
and
All crooks with identity less than the maximum value of the identities in the current set of PERSONS have been found
and
The SYSTEM is complete.

The intention is to express all predicates in terms of cardinal functions and transactions. This allows us to use the axioms and predicates of transactions to prove the correctness of a program, and it would be an important feature for an automatic proof-system. However, it also allows us to incorporate predicates as guards for correct execution of a program.

Example 3.1

The condition that the SYSTEM is ordered may be satisfied by the knowledge that all the files in it are sorted. However, we may also construct a transaction 'ordered?' which states the conditions for an ordered 'set_of_files' and which can be executed as a guard for the **do**-loop in (3.11). The best way to do this would probably be to construct a transaction 'still_ordered?' to test whether the 'current' component (see (2.14)) of a file is greater or equal than the previous 'current' component. This can be implemented as part of the 'read'-transaction for a file. It is left to the reader to construct the necessary transactions, and I will assume that {ordered?(SYSTEM)} can be used as a predicate.

Since 'complete?' and 'ordered?' are defined, the only informal part of the precondition (3.12) is the assertion:

"All crooks with identity less than the maximum value of the identities in the current set of PERSONS have been found."

This is the main invariant of the program 'find-crook'. As a matter of fact, the program might be regarded as a transaction to establish the truth of this condition after the end of one of the files in SYSTEMS has been reached. It is left to the reader to establish that this is an invariant of the **do**-loop in the following final version of 'find-crook':

```
(3.13) program find-crooks;
          use system file_investigation;
          use type CROOK_information;
          ident CROOK: person( CROOK_information );
          begin
              { defined?( SYSTEM ) }
                reset( SYSTEM );
                PERSONS := extract( SYSTEM );
                do { ordered?( SYSTEM ) }
                    complete?( SYSTEM ) ->
                        if same_person?( PERSONS ) ->
                                    "identify CROOK";
                                    move_equal( SYSTEM );
                         □ ¬ same_person?( SYSTEM ) ->
                                    move_less( SYSTEM );
                        fi;
                        PERSONS := extract( SYSTEM );
                od;
            end;
       end program;
```

The transactions 'move_less' and 'move_equal' reflects that the operation "move SYSTEM ..." as used in (3.11) depended on the value of 'same_person?(PERSON)':

- If **true** the maximum 'id' in PERSONS can be increased. Since all identifications are equal, we can move the SYSTEM by reading any or all of the component files.
- If **false**, we must try to make the smaller 'id's in PERSONS equal to the maximum. This can be done by reading any file for which the 'id' of the current record is less than the maximum.

The transactions, as defined below, should be included in the package (3.7) of 'set_transactions'.

```
(3.14) procedure move_equal;
          begin
            { ordered( sof ) }
              file.read( sof.f_1 );
              file.read( sof.f_2 );
              file.read( sof.f_3 );
            { ∀i, 1≤i≤3,
              | file.record( sof.f_i ).id > file.record'( sof.f_i ).id }
          end;
       procedure move_less;
          begin
            { ordered( sof ) }
              if less_1?( current ) -> file.read( sof.f_1 );
               □ less_2?( current ) -> file.read( sof.f_2 );
               □ less-3?( current ) -> file.read( sof.f_3 );
              fi;
            { ∃i, 1≤i≤3,
              | file.record( sof.f_i ).id > file.record'( sof.f_i ).id
          end;                                         '
```

At this point, the notations may be a little confusing. The dot "." is used both to indicate which structure a function belongs to (file.record), and to select a component of a record (sof.f_1 and file.record(sof.f_1).id). The apostrophe is used to indicate the previous 'record' of a file.

(3.14) introduces the transactions 'less_1', 'less_2' and 'less_3' on the 'current' set of records. With a proper indexing of the components of a record, this could have been reduced to one transaction. The transactions are defined in the package (3.10) of 'person_transactions'.

REFERENCES

(1) V.R. Basili and R.E. Noonan: "A Comparison of the Axiomatic and Functional Models of Structured Programming." IEEE Transactions on Software Engineering, Vol. SE-6, No. 5, September 1980.

(2) H. Boehm, A. Demers, J. Donahue: "An informal Description of Russell." TR 80-430, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, October 1980.

(3) E.W. Dijkstra: "A Discipline of Programming." Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1976.

(4) A.E. Fischer and M.J. Fischer: "Mode Modules as Representations of Domains. Preliminary Report." Massachusetts Institute of Technology.

(5) J.V. Guttag: "Abstract Data Types and the Development of Data Structures." Communications of the ACM, Vol. 20, No. 6, June 1977.

(6) J.V. Guttag: "Notes on Type Abstraction." IEEE Transactions on Software Engineering, Vol. SE-6, no. 1, January 1980.

(7) J.V. Guttag and J.J. Horning: "The algebraic Specification of Abstract Data Types." Acta Informatica, Vol. 12, No. 10, 1978.

(8) E. Horowitz and S. Sahni: "Fundamentals of Data Structures." Computer Science Press Inc., 1976.

(9) J.D. Ichbiah, J.C. Heliard, O. Roubine, J.G.P. Barnes, B. Krieg-Brueckner and B.A. Wichman: "Rationale for the Design of the ADA Programming Language." SIGPLAN Notices, Vol. 14, No. 6, Part B, June 1979.

(10) K. Jenssen and N. Wirth: "Pascal User Manual and Report." Springer-Verlag, New York, 1974.

(11) A. Laut: "Safe Procedural Implementations of Albebraic Types." Information Processing Letters, Vol. 11, No. 4,5, December 1980.

(12) C.H. Lewis and B.K. Rosen: "Recursively Defined Data Types, Part 1." IBM Thomas J. Watson Research Center, Yorktown Heights, New York.

(13) R.L. London, J.V. Guttag, J.J. Horning, B.W. Lampson, J.G. Mitchell, and G.J. Popek: "Proof Rules for the Programming Language Euclid." Acta Informatica, Vol. 12, No. 10, 1978.

(14) J.H. Morris: "Types are not Sets." ACM Symposium on the Principles of Programming Languages, pp. 120-124, October 1973.

(15) D.R. Musser: "Abstract Data Type Specification in the AFFIRM System." IEEE Transactions on Software Engineering, Vol. SE-6, No. 1, January 1980.

(16) R. Nakajima, M. Honda and H. Nakahara: "Hierarchical Program Specification and Verification - a Many-sorted Logical Approach." Acta Informatica, Vol. 14, No. 2, 1980.

(17) D.L. Parnas: "On the Criteria To Be Used in Decomposing Systems into Modules." Communications of the ACM. Vol. 15, No 12, December 1972.

(18) L.A. Rowe: "Data Abstraction from a Programming Language viewpoint." Computer Science Division-EECS, University of Califirnia, Berkeley, CA 94720, 1980

(19) J.W. Thatcher, E.G. Wagner and J.B. Wright: "Data Type Specification: Parameterization and the Power of Specification Techniques." IBM T.J. Watson Research Center, Yorktown Heights, NY 10598

(20) E.W. Williams, Gen. Ed.: "The Scribner-Bantam English Dictionary." Bantam Books, Inc., 1979.

(21) T. Winograd: "Beyond Programming Languages." Communications of the ACM, Vol. 22, No. 7, July 1979.

(22) N. Wirth: "Program Development by Stepwise Refinement." Communications of the ACM, Vol. 14, No. 4, April 1971.

(23) N. Wirth: "Modula: a Language for Modular Multiprogramming." Software - Practice and Experience, Vol. 7, 1977.

(24) N. Wirth: "MODULA-2." Institut fur Informatik, ETH, Zurich, March 1980.

(25) N. Wirth: "Algorithms + Data Structures = Programs." Prentice_Hall Series in Automatic Computation, Englewood Cliffs, New Jersey, 1976.

(26) W.A. Wulf, M. Shaw, P.N. Hilfinger and L. Flon: "Fundamental Structures of Computer Science." Addison-Wesley Publishing Company, 1981.