DATA TRANSMISSION AND MODULARITY ASPECTS OF PROGRAMMING LANGUAGES

by

Arndt von Staa

Research Report CS-74-17

Department of Computer Science

University of Waterloo
Waterloo, Ontario, Canada

October 1974

INDEX

ACKNOWLEDGEMENTS

I wish to express my deep appreciation to my supervisor, Professor W.M. Gentleman, for his constant guidance, suggestions and criticisms made during the writing and research of this dissertation.

I wish also to extent my gratitude to Professor D.D. Cowan for his encouragement and moral support during my stay at the University of Waterloo.

The financial support of the Pontifícia Universidade Católica do Rio de Janeiro, Brazil, is gratefully acknowledged.

ABSTRACT

In this dissertation we will study the problems relative to the interchange of information between program modules. Although the results apply also to information transmission between machines, we are primarily interested in developing techniques such that several program modules may interchange information in a meaningful way, whether or not these modules belong to the same program, whether or not they have been independently compiled and whether or not their corresponding source code applies to different language processors.

As well as the problems related to information interchange, we will also study how to build modules in such a way that modules which produce the same computation are interchangeable, that modules may be independently compiled and that modules are adaptive to the program in question in the sense that they do not have to be manually encoded again in order to fit into this particular program.

We will devote our attention to (i) data types, (ii) how information is accessed and protected from unauthorized accesses, (iii) how modules interchange information, and (iv) how modules behave in the presence of run-time detected exceptions.

1. Introduction.

1.1 Statement of Goals.

In this dissertation we will study the problems relative to the interchange of information between program modules. First let us examine the relevancy of such a study.

In today's engineering practice the use of building blocks is becoming more and more widespread. The main driving forces being economy due to the repeated use of a same block, and speed of design due to the decrease of the level of design detail. Building blocks exist in current programming languages in several forms, such as functions, procedures, macros etc. Usually though, they lack of flexibility and/or accuracy of composition, where the latter means that language processors seldomly perform a validation of the data paths between modules. The design of tools, or better of language constructs which allow the construction of flexible and accurately composable modules is thus badly needed in today's programming practice. Our goal is then to design tools so as to produce modules which are:

a- flexible in the sense that a given module may serve for several different applications. For example, a module implementing a stack should not freeze the type of this stack until this module is effectively used. This would allow the same module to be used in several different places, e.g. as a stack of integers, or a stack of module instances etc.

b- composable as defined by Dennis[Den3]. That is, it should be possible to build modules from other modules without any restriction of hierarchical depth.

c- accurately composable in the sense that, when composing modules, the data paths are examined for meaningfulness.

d- non-interfering but through well defined data paths [Den3]. This implies that the amount of information which can be interchanged between two modules should be kept minimal[Mye1].

e- diagnosable in the sense that exceptions detected during run-time are either recoverable or produce descriptive information at the source language level rather than at hardware level.

It is a natural consequence of this statement of objectives that we should answer following questions:

a- what is a data type? This question must be answered, since we desire to impose a meaning on the information being interchanged, where this meaning conveys more information than just a description of the set of possible values [chapter 3].

b- how is information made available and/or accessed? This question must be answered, since we desire to minimize the amount of information a module makes available and/or reqires, although we do not want to impose restrictions on the class of programs which could be written [chapter 4].

c- how are data paths established between modules? This question must be answered, since we must be able to define how modules interchange information in a meaningful way [chapter 5].

d- what actions are to be taken when exceptions are detected at run-time? This question must be answered, since exceptions could be valid information in the sense that they may drive the elaboration of a program and, consequently, may be interchanged between modules [chapter 6].

In order for such a study to be of practical value, we may not solve problems by imposing restrictions on the form or content of programs which could be written. Since we are unable to foresee all possible applications, this amounts to not restricting the power of expression of the language.

It should be emphasized that we are not proposing a new programming language. Rather we are studying the semantical aspects of the constructs required in order to achieve the stated goals. As a consequence we have not attempted to define a syntax. Instead, we borrow notation from several existing programming languages, more notably ALGOL 68[Win1,Lin1] and PASCAL[Wir2,Wir4].

This study is not complete, i.e. not all problems relative to modularity and information interchange are examined. For instance, we are not dealing with problems related to the existence of multiple access paths to a same data space. The reason for this incompleteness is that it seemed to be quite natural to break the study at a point where dynamic interference began to appear more prominently.

1.2 Overview of the Dissertation.

In chapter 2 we examine the concepts relative to storage and the use of storage within programs written in a symbolic language. The main purpose of this chapter is to define basic concepts and terminology which will be used throughout the remainder of this dissertation.

In chapter 3 we will study the meaning of data objects i.e. types. Following Morris[Mor2] we do not consider a type as being just the definition of a set of values, but, rather a type defines both a set of values and a set of operations on these values. Thus, our concept of type resembles the concept of data structure[Ear1,Ear2,Fle1,Har2, Hoa1,Hoa2,Knu2,Sta1,Tur2,Weg2] rather than a simple space descriptor as, for instance, structures in PL/1. From the implementational point of view, our types can be viewed as being similar to classes in SIMULA67[Dah3] or to clusters as defined by Liskov and Zilles[Lis1].

It follows from our definition of type, that it makes sense now to talk about types such as: "stack of user type", "string of user type", "buffer of user type" etc., where these types are characterized primarily by the operations which can be performed on them. We will also be able to define types such as: "prime integers", "square upper triangular matrices" etc., where these types are characterized by being a proper subset of some well defined basic type, such as the integers or square matrices respectively.

The first advantage of generalizing the concept of types, is that we will be able to talk openly about the properties of a given type. That is, we are no longer

obliged to hide these properties in code which is sometimes difficult to understand. It does contribute then to:

a- understandability of programs, or, as sometimes called, structuring of programs;

b- interchangeability of implementations, this follows from the fact that the implementations are greatly transparent. Thus we are able to chose the best suited implementation of a given type without affecting other portions of the program.

c- modularity, that is, programs can be broken into several small programs, where the interaction between these programs occurs through clear and well defined interfaces.

Most of the concepts are based on, and extend, concepts introduced by the SIMULA67 class mechanisms[Dah2, Dah3,Dah4,Hoa4]. Several ideas were also taken from work done by Hoare[Hoa5], Wirth[Wir2,Wir4] and Parnas[Par2, Par3].

From the point of view of inter-module transmission, this chapter is important mainly because we want to assure that programs receive the data they expect. In this case it becomes even more necessary that we do not just pass unqualified "integers" for instance, but that we are able to pass a "prime integer". If we observe the techniques used conventionally, what we are proposing is nothing more than to mechanize part of what is done verbally or through documentation, i.e. programming discipline.

In chapter 4 we study how information is accessed and/or made available. Internal spaces of some module M, e.g. a procedure or type descriptor, may be accessed by means of

some access function defined by M or global to M. Such access functions, or locator functions, could occur in a variety of forms and degrees of complexity. For example, some access functions could be made responsible for traversing a tree in some predefined order. Others could be made responsible simply for accessing dynamic storage spaces. Finally, others could perform checking operations in order to determine whether a given procedure may effectively access the data space in question. We may say then, that access functions serve at least for the following two purposes:

i- to characterize the storage behaviour of a given type, e.g. tree, stack, buffer etc.;

ii- to protect the data spaces internal to the modules defining the access function.

Access functions, or locator functions, could take the degenerate form of just loading an address into some register. Other access functions could compute such addresses by means of some user provided parameter. Finally, others could receive or establish control information from which these addresses could be abstracted.

When performing list processing, usually some pointer, or reference, is used to refer to a particular element of the list. It has been recognized though, that the generality of pointers may cause the certification of programs to become quite difficult[Hoa7]. This suggests then, that an alternative for pointers be found, where this alternative should:

a- not restrict the power of expression of the programming language, nor cause the textual dissociation of related entities;

b- restrict as much as possible the set of data spaces able to be accessed;

c- hide as much as possible the implementation of modules which make internal spaces available to the exterior;

d- allow verification of access rights.

We will show in this chapter that these objectives can be met. As a result we show that several language constructs become necessary. However, we will not study whether some of these constructs are necessarily inefficient.

In this chapter we will also examine how successive elements of a given ordered set could be generated. It should be noted that these sets or, better, sequences could be defined in a computational manner, e.g. the sequence of nodes visited when traversing a tree in some order. We will examine then the constructs which are necessary in order to implement generator functions. The concept of generator functions has been defined for a long time already, e.g. IPL-V[New1,Gel1]; it has received little consideration in present programming languages though. It will be shown that by means of generator functions the implementational characteristics of a module could be hidden, thus contributing significantly to the interchangeability and flexibility of modules.

Another problem which arises when information is interchanged between modules is that of privacy and access path control, e.g. reference count. It will be shown that, in order to overcome these problems, we must be able to redefine or extend language processor defined operators, in particular the store operator. Of course, these access con-

trols are weak in the sense that the language conventions must be obeyed, since, otherwise, the controls become ineffective.

In chapter 5 we will study the problems relative to information interchange between modules. In particular we will examine how independently compiled modules may interchange information.

Information may be interchanged in several ways, e.g. by means of global areas, actual/formal parameter lists as well as by means of message transmission. We will show that, from the point of view of actually making information available, we can study these three different forms of information interchange in a unified manner.

The basic concept which we will use for interchanging information is the parameter list. We will define parameter list association in such a way that it becomes explicit which parameters are to be associated. As we shall see, this allows us to piecemeal associate parameter lists. It also allows us to dissociate the textual order of parameters from the order in which parameters occur in parameter lists. That is, we will depart from the positional association rule of parameters which is common to most of the present day languages. Finally, due to our parameter list association rule, it becomes possible to define one single parameter list which contains formal, global as well as message (sender/receiver) parameters.

Besides parameter lists, we will also study module typed variables and the activation forms of modules. Thus, we will study control flow and data flow driven modules. Furthermore, we will relate interrupt handlers with data flow driven modules.

Module typed variables pose several problems of their own. We are particularly interested here in defining association of parameter lists in the presence of module typed variables. As we shall see, such module typed variables imply that association will act upon parameter list typed variables. Furthermore, module typed variables imply the need for template parameter lists in addition to actual and formal parameter lists. Such a template parameter list describes the parameter list related to the uses (calls) of module typed variables.

Module typed variables will be treated in the same way as any other variable. It is thus valid to define arrays of modules or functions which return module typed values.

Finally, in chapter 6 we study exception handling. There are several reasons why such a study is important:

a- programs might be interrupt driven. For example, interactive systems, e.g. time-sharing systems, usually allow executing programs to be preempted and later to be resumed by means of user interaction.

b- some operations may fail and this failure may direct further action, where such a failure is not to be regarded as an error. For example, when reading records from a sequential file, e.g. tape file, a read request may fail due to the file having been exausted. In this case a summary submodule is frequently started.

c- a given module M could define several submodules, each of which attempts to solve the same problem in a different way, e.g. using different algorithms and/or starting values. By means of successive trials of

these submodules and/or starting values, the module M could eventually produce the desired result.

d- machine failures could cause program elaboration failures. Such failures are non-avoidable since every machine possesses a (usually very slim) probability of malfunction. Notice that such failures occur frequently in input/output handling modules and appropriate safeguards are incorporated into these modules.

e- the program itself may contain errors and thus reach abnormal states. Ideally programs should be proved correct[Dij4,Flo1,Hoa2]. Pragmatically, though, this ideal cannot be achieved when using the tools currently available[Hor1]. Thus even when programs have been "proved" correct, they may still be incorrect[Sch4] possibly due to incorrect proofs, misunderstanding of the program environment, unforeseen conditions etc.

From items (a) through (c) it folows that we may desire to use exceptions as a tool for solving a given problem. From items (d) and (e) it follows that we must be able to cope with unexpected exceptions in order to prevent serious damages to the program (or system) and to produce information which, hopefully, aids fault diagnosis.

We will study in this chapter how exception conditions can be defined so as to allow user defined conditions and parameters to exception handlers. Since exception handlers can be equivalenced to data flow driven modules, as will be shown in chapter 5, we are in fact studying here the tools necessary to implement data flow driven modules.

The point where an exception is detected is not necessarily within the module instance which defines the corresponding exception handler. We must examine then how a detected exception may be passed from module instance to module instance in order to be serviced eventually. We will examine also the ways in which control can be given back to the module instance where the exception was detected.

Due to the parallel or quasi-parallel nature of exception handlers, there are several timing problems which must be examined. Our study will concentrate in determining the different timing problems and how they could be overcome. However, we will not study how deadlocks could be detected and/or prevented.

2. Basic Concepts Regarding Storage.

In this chapter we will study concepts relative to storage and the use of storage within a program written in a symbolic language. Furthermore, we will consider only the aspects of unstructured storage.

In section 2.1 we define space, names, access paths and textual names. In section 2.2 we define several forms of scope. In section 2.3 we list the formal definitions of the more important concepts introduced in sections 2.1 and 2.2.

2.1 Space, Names and Textual Names.

A storage **medium** is a device where information may be placed, and/or from where it may eventually be retrieved. Examples of media are: core; magnetic tapes, disks and drums; punched cards; output forms.

A piece of information occupies a portion of the medium where it is stored. This portion is characterized by a starting position within the medium, i.e. **address**, and by an **extent**. Examples of addresses are: core address; the tape drive designation and the position of the tape on this unit. Examples of extents are: word lengths and record lengths.

The storage space characterized by the triple <medium, address,extent> will be called **space**. Notice that the concept of space presented here is completely void of structure and interpretation [i.e. type]. Furthermore, the way in which space has been defined implies contiguous sections of storage only. In chapter 3 we will study how to

attach meaning to spaces, and also how different spaces may become interdependent.

The next important point with regard to spaces is that they are viewed at the level the user sees the machine. That is, if the hardware allows the implementation of virtual memory[Ben2,Bob2,Den1], the spaces within the main store are placed on the medium "virtual memory". This medium appears then as if it were homogeneous, although in reality it is not. This may affect the behaviour of the program, in particular with regard to efficiency[Coh1, Hat1].

Finally, there may be several devices containing a certain kind of medium, e.g. magnetic tape units. In these cases the addresses will be composite in that they define both the unit and the location of the information within this unit.

When executing a program, spaces will be accessed either to retrieve data, i.e. read access, or to save data for eventual use, i.e. write access. In order to decide which space the program will access, the program must first obtain a value which defines the triple <medium,address, extent>. Such values will be called names. Since names are values, they also require storage space in general. In order to break the infinite recursion generated in this form, we need the concept of implied names. Implied names are established by convention in such a way that there is no need to keep explicit information for this kind of name. For example, the binary operator in a Burroughs B6700, accesses, by convention, the topmost two elements in the current execution stack†[Bur1,Hau1,Org2,Org3]. Thus by

---

† We have simplified the access form of the B6700 for reasons of simplicity only. In fact more complicated access forms, such as indirect references, are possible.

convention when a binary operator is decoded, the medium of both operands are the stack, the addresses are the topmost and the next to the topmost stack elements, and the extent is the word length. Consider now the case of a single operand [address] machine, e.g. Honeywell 6000, IBM 1130, IBM 7090. A binary operator in such a machine implies the accumulator register as one of the operands. The other operand is defined by the core address defined by the instruction. For reasons of simplicity, let us assume that there is no indexing and/or indirect addressing available. In this case the implied name would be <instruction register, address portion start, address length>. Observe that this is the name of a space from which the name of the operand space is effectively retrieved.

In the same way as implied names define by convention all three fields of the space characterizing triple, there may be conventions which imply the definition of one or more fields of a name value. For instance, in many machines instructions imply core as the medium. Also in several machines the extent is predefined to be a word length.

We will call <u>space allocation</u> the operation of obtaining a space on some medium. Notice that this operation corresponds to the creation of a name value, or of a set of name values. To perform this operation a name value is also required, since we need to know of the existence of the space being allocated. Thus, in fact this operation does not correspond to an effective "creation" of space, but just a means to give the user a name value to a space he did not already possess. Observe that space allocation does not imply any data transmission to the allocated space. Conversely, <u>deallocation</u> is the operation of relinquishing a space. Again, this operation just serves

for the purpose to take the space away from the set of spaces dedicated to the user. An immediate consequence of deallocation, is that any name value characterizing a deallocated space is no longer a valid name value. In chapter 4 we will study the operations of allocation and deallocation with more detail.

Names may be computed at run time. For example, when adding [or subtracting] index registers and when performing indirect referencing. The values used to perform the computation may even be defined by the space being accessed [e.g. the B6700 for indirect references]. Name values used to access some space, will then be obtained by a series of operations, i.e. by means of a function. This function will be called *access path*.

From the discussion in the preceding paragraph we may observe that the information contained in a space is subjected to some interpretation, i.e. *type*. Observe that we already mentioned name typed values and instruction typed values. We will leave the detailed discussion of types to chapter 3.

The operation which associates a certain name value with a given access path for a given set of parameters will be called *binding*. We will use this term despite the fact that it is used by some authors with a slightly different meaning. For example, Watson[Wat1] uses the term binding as an operation which associates a virtual address with a real address within virtual memory machines. In the MULTICS system[Dal1,Org1], the term is used to denote the replacement of external references by actual addresses [the make known process]. Notice that binding in our definition associates a name value with one element of the cartesian

product of access paths and parameters to acces paths. However, this does not imply that a set of such elements cannot be bound all at the same time. For example, when defining the starting address of an array, usually the whole array is bound at once.

We have to emphasize at this point, the difference between space allocation and binding. The space allocation operation basically creates a valid name value. On the other hand, the binding operation enables an access path to compute a, hopefully valid, name value.

Binding is not necessarily done all at one single instant. For example, when using Dijkstra's[Dij1] approach to implement ALGOL60, we have partial binding occuring at compile time. The displacement of local variables from the block start can be defined at compile time. However, the position of the block start is defined at run time. Binding of this kind will be called partial [static] binding. Compile time† binding will be called static binding. For example, FORTRAN IV non parametric variables are statically bound in most implementations. Run time binding will be called dynamic binding. For example, FORTRAN IV parameters passed by reference are an instance of dynamic binding.

In the case of indirect addressing we have several levels of binding, since we must provide bindings from each indirect address word to the corresponding space in the indirect chain.

Since access paths are in fact computations, they may deliver wrong name values. This may occur due to an improper use of the access path, or due to the nonexistence of binding. From this we have that there should be a

---

† Notice that loading is an instance of compile time.

special name value, say not_bound, which flags this latter case. Thus any evaluation of an access path which delivers not_bound results in an error condition. Evaluation of access paths may also fail during the evaluation, i.e. before even delivering a name value. We have then the following error conditions associated with access paths:

i- attempt to use the "space" denoted by not_bound;

ii- improper use of the access path, e.g. parameter errors.

When programming in symbolic languages, we do not refer to spaces by means of name values. Rather we use symbols, i.e. textual names, which stand for such name values. There is then a need for an operation which transforms a textual name into an access path. This operation, i.e. function, will be called name map. Observe that the name map is a "function value" delivering function.

Examples of textual names are:

a- POINTER->BASED_ARRAY_A(I,J) in PL/1

b- $('Name' STRING) in SNOBOL4

c- <<PROJECT_DIRECTORY>COMPILER1 in MULTICS file system

In several programming languages a given textual name may refer to several different spaces. This may occur due to the ability of generating several different access paths or, then, due to the associated access path being capable of delivering several different name values. For example, local variables in recursive procedures generate one access path capable of accessing several different spaces. Since we want to achieve deterministic results, each successful

evaluation of an access path must deliver exactly one name value. Furthermore, each successful evaluation of a name map must produce exactly one access path. Thus, there must be two sets of parameters, each one sufficient to define explicitly which of the several values is to be chosen. Of course, some of these parameters could be implied by the programming language or its implementation.

Our definition of textual names is such that, for instance an object name in SIMULA67 is not a textual name, unless qualified by an object reference. This follows from the object name being unable to define a unique access path, unless it specifies which of the many objects is being refered to.

We could argue that there is no distinction between access paths and name maps. For instance, a construct like $('ABC' FUNCTION(I)) in SNOBOL4 is a textual name. No work can be done by the name map though, since we do not know what is being accessed, i.e. the effective textual name. Observe that this construct contains several other textual names, e.g. 'ABC', FUNCTION(I), I. This example also shows that textual names are in fact values and that they could be computed at run time, e.g. the result of ('ABC' FUNCTION(I)).

Textual names could be created and/or erased in a non-computational way. For instance, in APL\360 a programmer may suspend execution and later request the resumption of the execution of some program. During the suspended state, textual names may be created and/or deleted [)ERASE, see Pak1].

We will call <u>locator function</u> the composite function corresponding to the name map and the access path delivered

by a given evaluation of this name map. Locator functions will be studied in detail in chapter 4.

2.2 Scope of Names.

Let A be a textual name. The locator function of A, say L, may deliver several name values, say $a_1, a_2, \ldots, a_n$. This set of possible access paths may vary at run time. We will call dynamic scope of A relative to a_i, the time interval during which the textual name A is capable of representing a_i. We will call A-activation of a_i, the operation which associates a_i with A, i.e. the definition of the beginning of the dynamic scope. The operation A-deactivation of a_i, is the operation of relinquishing the association of A with a_i. Observe that the dynamic scope is a continuous time interval. Observe furthermore, that the dynamic scope could possibly be the duration of execution of the whole program, as for instance static storage in PL/1, or non parametric variables in FORTRAN IV.

Our definition of dynamic scope differs slightly from the usual definition. Usually dynamic scope is defined as being the set of time intervals during which a given textual name A stands for some specific name a_i. We have chosen a different definition for dynamic scope, in order to explicitate the existence of a selection procedure which selects one of a possible set of names associated with a given textual name. A further consequence of our different approach to dynamic scope is that we establish a correspondence between the dynamic and the textual scope aspects. This will become apparent later in this section.

Notice that binding is an instance of activation. This shows also that, in the presence of name typed values, there may be several dynamic scopes of a textual name A relative to the same name a_i. For instance, a pointer to a node in a list may point to the same node at different time intervals, without being restricted to point to this node during intervening intervals.

Similarly we will call dynamic scope of a space α, the time interval starting at the allocation of α, and ending at the deallocation of α.

There must be several parameters which specify which of the many name values is the one required, say a_i. We will say then that all name values a_j, $i \neq j$, are in the supressed scope with respect to these parameters. The importance of suppressed scope arises when implied parameters are used. For example, all instances of a local variable in a recursive procedure, except for the most recent one, are in the supressed scope. The operations of suppression and reactivation, are the operations which, respectively, define the starting and the ending of the suppressed scope.

To clarify some fine points, let us consider the following examples:

a- the dynamic scope of a non-parametric variable in FORTRAN IV is the whole duration of the program's execution.

b- the starting and termination of the execution of a block in a block structured language, delimit a dynamic scope instance of the local variables defined in this block. Observe that a new execution of the

block defines a new dynamic scope of a new set of spaces. Thus, in a recursive succession of activations, suppression will occur.

c- the dynamic scope of a natural variable in SNOBOL4 is initiated by the definition, or redefinition, of this particular variable. It lasts until a further definition or redefinition of this same variable, or, then, until program or local instance end. Suppression may occur if a function for which this variable is local is activated. Suppression may also occur if the value of the variable is required after an apparent deactivation of the variable. For instance, if a user defined data type is redefined, the old definition must be kept until there is no more data using this old definition.

We will call dynamic reach of a textual name A relative to a name a_i, the set of time intervals of the dynamic scope of A relative to a_i, during which a_i is not in a suppressed scope. Notice that this definition of dynamic reach is identical to the conventional definition of dynamic scope mentioned earlier in this section.

With regard to the relationship between text and textual names, similar problems may occur. We say that textual scope is the portion of text within which a given textual name may be used. Furthermore, textual activation and textual deactivation are text operations which, respectively, delimit the starting and ending of the textual scope. Observe that textual scope defines several continuous portions of text, i.e. it does not imply that there is only one instance of textual scope. Consider for instance FORTRAN IV implementations where declarative

statements must precede the executable statements of a program unit. The names of COMMON areas are then confined to these portions of text, although a certain name may occur in several of these text portions.

In some programming languages a same textual name may have several different meanings at a same textual point. For instance, we may redeclare a global variable in ALGOL60; in FORTRAN IV the same variable may be used in several program units, or may have different meanings depending on context, e.g. COMMON name and variable names in FORTRAN IV; a textual name may represent a label, a function and a value, all at the same time in SNOBOL4. Again we need to provide parameters, explicit or implicit, in order to uniquely specify which of the several possible meanings we desire. We will say then, that all other meanings are textually supressed with respect to this set of parameters. Observe that this is done frequently by qualifying textual names by hidden parameters, such as: "historic block definition" in ALGOL60; "program unit" in FORTRAN IV; "user id" in Honeywell 6000 files. The operations defining the start and the end of the textual suppressed scope are textual suppression and textual reactivation respectively.

Following the nomenclature of ALGOL68, we will call textual reach the section of code within which a given textual scope is textually active and not textually suppressed. Observe that textual reach is not necessarily a fixed portion of code. For example, a language could be designed where portions of code are inserted and/or deleted. Thus, scope rules could be affected at run time. Notice that such a language could be built within the

SNOBOL4 environment. This is due to the fact that SNOBOL4 allows run time compilation by means of the CODE function.

We will deal at many points with the concept of program module. We will thus anticipate its definition at this point. A program module, or simply module, is a contiguous portion of source text, such that the textual scope of the textual names used by this program module completely contains, or is completely contained by, the text of this module. For example, ALGOL60 blocks are modules; FORTRAN IV program units are modules. However, only a complete SNOBOL4 program is a module. This does not imply that we could not program in a modular way in SNOBOL4, it only says that there is no natural definition of module in SNOBOL4. Notice that using this definition, macros could be viewed as modules.

2.3 Summary of definitions.

Defn. 2.1 Space is a portion of storage on some medium, starting at some address and occupying some extent of storage space. It is characterized by the triple <medium,address,extent>.

Defn. 2.2 Name is a value of the form <medium,adress,extent> which refers to the space characterized by this triple.

Defn. 2.3 An access path is a computational process, i.e. function, which delivers name values for the purpose of accessing the corresponding space.

Defn. 2.4 Binding is the operation which associates a name

value with a given access path and a given parameter value, or set of parameter values.

Defn. 2.5 Textual name is a character string with which the programmer refers to spaces in a symbolic programming language.

Defn. 2.6 Name map is a computational process, i.e. function, which maps a textual name onto an access path.

Defn. 2.7 Dynamic scope of a textual name relative to a name value, is the time interval during which this textual name is capable of representing this name value.

Defn. 2.8 Textual scope of a textual name, is the portion of text within which this textual name has a specific meaning.

Defn. 2.9 A program module is a portion of source text such that the textual scope for all textual names used by this program module is either completely contained by this program module, or, then, completely contains the text of this program module.

3. Types.

In this chapter we will study the data objects as such. Usually the contents of a given space are not just a sequence of bits, but it possesses a definite meaning. This meaning, i.e. type, is our object of study here.

We have chosen the term type, rather than mode, as in ALGOL68, or attribute, as in PL/1, simply because it seems to be known more widely.

Our concept of type is a generalized concept. It resembles more the concept of a data structure, than that of a simple space descriptor, as for instance, structures in PL/1. That is, a type not only specifies the lay out, but it also specifies the set of operations and the set of valid values of this type. Thus it makes sense to talk about types such as: "stack of user type", "string of user type", "buffer of user type" etc., where these types are characterized primarily by the operations which can be performed on them. We will also be able to define types such as: "prime integers", "square upper triangular matrices" etc., where these types are characterized by being a proper subset of some well defined basic type, such as the integers or square matrices respectively.

The first advantage of generalizing the concept of types, is that we will be able to talk openly about the properties of a given type. That is, we are no longer obliged to hide these properties into code which is sometimes difficult to understand. It does contribute then to:

a- understandability of programs, or, as sometimes called, structuring of programs;

b- interchangeability of implementations, this follows from the fact that the implementations are greatly transparent. Thus we are able to chose the best suited implementation of a given type without affecting other portions of the program.

c- modularity, that is, programs can be broken into several small programs, where the interaction between these programs occurs through clear and well defined interfaces.

Most of the concepts are based on, and extend, concepts introduced by the SIMULA67 class mechanisms[Dah2, Dah3,Dah4,Hoa4]. Several ideas were also taken from work done by Hoare[Hoa5], Wirth[Wir2,Wir4] and Parnas[Par2, Par3].

From the point of view of inter-module transmission, this chapter is important mainly because we want to assure that programs receive the data they expect. In this case it becomes even more necessary that we do not just pass unqualified "integers" for instance, but that we are able to pass a "prime integer". If we observe the techniques used conventionally, what we are proposing is nothing more than to mechanize part of what is done verbally or through documentation. We will study the advantages of doing so and, mainly, the limitations imposed.

In section 3.1 we provide the basic definitions regarding types. These definitions will be used throughout the remainder of this dissertation. In section 3.2 we discuss how types are associated with spaces. We also introduce the concept of type conversion, or data transfer, in this section. In section 3.3 we will study type descriptors as if they were computational values. That is,

we will study the basic operations which can be performed on values of type type. In this section we study also the problems relative to the dissemination of type descriptors among subsections of the program. In section 3.4 we will study how to identify type descriptors, and how we could define equality of types. Finally, in section 3.5 we will study how type-wise correct accesses could be enforced, both dynamically and statically.

The concepts studied in these sections are highly interdependent. This constitutes a major difficulty when attempting to partition these concepts into a nice succession of ideas. We based our partition choice on an attempt to minimize the number of times where we require concepts which have not yet been defined.

3.1 Types and Spaces.

When studying spaces in chapter 2, we mentioned that the information carried by a space, i.e. the values of the bits in this space, is subject to some interpretation. This interpretation may be due to the use of an operator, e.g. the integer add instruction A on an IBM/360[IBM1]; the reference operator "." in BLISS[Wul1,Wul2]. The interpretation may also be provided by the space itself, e.g. a Burroughs B6700[Bur1,Org2,Org3] data word defines the type of word [stack marker, indirect reference etc.]; the contents of a SNOBOL4[Gri6] variable defines its own type. Finally, the set of spaces may be partitioned into several sets, each of which contains one only type of data, e.g. ALGOL68[Win1,Lin1] variables; PASCAL[Wir2,Wir4] variables. Furthermore, we have only considered contiguous spaces in chapter 2. However, depending on the interpretation used,

several spaces may contain interrelated information under this interpretation.

Defn. 3.1.1 A space set $A=\{\alpha_1,\alpha_2,\ldots,\alpha_n\}$, is a collection of zero or more spaces.

Defn. 3.1.2 The type of a space set A, is the interpretation given to the information carried by A.

The ordered couple $\langle A,\tau\rangle$, where τ is the type of the space set A, will be called data space. Observe that a data space may be built up from zero spaces. In this case there is no information in the data space. Such data spaces will be called null spaces, and their corresponding type will be called a null information type, nit for short. Notice that there may be several nits in a given program.

The most widely used data spaces are those where the underlying space sets contain exactly one element, i.e. space. For example: integers, reals, vectors of integers. An example of data spaces where the underlying space set contains more than one element is a list in LISP1.5[McC1], or a slice of an array in ALGOL68.

Data spaces may contain subspaces which in turn are data spaces. For instance, the elements of a "vector of reals" are "reals". This shows that a given type may have subtypes and that we may build types by logically associating several types. In section 3.3 we will study operations on types in greater detail.

Several types may be defined by a given machine. When implementing compilers, user defined or system defined types must be mapped somehow onto a set of types made available by the host machine.

Defn. 3.1.3 A primitive type is a type for which there are valid single instructions in the host machine.

A primitive type is not necessarily defined over one space only. For example, in a segmented paged machine, the type "virtual address" is defined over a set of spaces, e.g. segment table and page tables. A primitive type is also not necessarily indivisible. For example, in an IBM/360, there are four characters, i.e. 8 bit bytes, per word. Finally, there may even be smaller spaces, in number of bits, than the smallest addressable primitive type. For example, a PL/1 BIT(1) string may be implemented in an IBM/360, although the smallest addressable unit is a byte of 8 bits.

Having in view that higher level languages represent just some virtual machine, we may also have language primitive types. That is, such types for which there are valid single operators defined within the language, even if just fetch and store, and which are not structured. E.g. BIT(1), CHAR(1), PASCAL scalar types, but not strings, arrays or complex. We have then, that a simple type is a primitive type, a null information type or an unstructured language primitive type.

Defn. 3.1.4 A composite type is a logical association of zero or more types, each of which is either a simple type or a composite type.

If we had not defined bits as simple types, things like BIT(1) in PL/1 would necessarily be composite types. This would introduce an infinite recurrence in the above definition, since BIT(1) does not terminate the recurrence of the definition. Rewording the definition would also be of no help, for then we would have to define composite

types in terms of composite types. Observe that, for n≠1, a bit string BIT(n) in PL/1 is necessarily a composite type due to the definition of simple types. This suggests then, that a bit string is an array of type bit. Similarly a character string is an array of type char.

Consider now name typed spaces. Such spaces may contain things like: pointers [references], labels, procedure names, file names, remote format names [e.g. FORTRAN IV, PL/1], etc. There are always two interpretations for such a kind of data space:

a- the data space containing the name typed value is in itself the space being considered.

b- the space characterized by the name value contained in this data space is the space in question. In this case, the type of the data space being considered, is the one implied by the name and/or by the space itself.

Observe that the interpretation (b) occurs only when the name typed data space is given to a locator function as a parameter. This is so, since only locator functions use name values to gain access to the space characterized by that particular name value. This justifies following convention:

Convention 3.1.5 Outside the scope of locator functions, name typed spaces will always be interpreted as a value of type name, and not as the space characterized by this value.

This convention implies that we may study composite types containing names as subtypes, as simple entities without regarding the spaces refered to by these subspaces.

It is worthwhile to notice, that the process of constructing a locator function lies outside the scope of this refered locator function.

We must stress here the fact, that a locator function is a mapping from a textual name onto a name, i.e. data space. Thus, the "dereferencing" operation [coercion] in ALGOL68 is in fact part of the locator function of a textual name of type ref ref real, i.e. pointer to real.

Type definitions have a very strong influence on space allocation. This follows from the fact that several language processors will create space allocation requests based solely on type definitions. It follows then, that there could be some type definition which would not allow such a creation to complete. For example, following "ALGOL68" construct:

```
type A=union(integer;
             type B=struct(integer b; A c));
```

defines the types (mode) A and B. Type B is defined in terms of type A, and type A in terms of type B. Suppose this construct would be used not only to define a type, but also to define the amount of space and the internal layout of a data space bearing this type. We could then easily verify that this type definition cannot possibly generate a space allocation request. This follows from the fact that we do not know, a priori, how many recursion levels are to be taken. On the other hand, it can be verified that a sequence of creations of values defined in this way do not imply any major difficulties, but for implementational ones. The same is also pointed out by Lewis and Rosen[Lew1].

```
A r,s,t;
      r:=1; s:=(2,r); t:=(3,s);
```

generates following values:

r=1 s=2 (1) t= 3 (2 (1))

where (x) means: x occurs as a type A value.

Figure 3.1.1 A sequence of creations of data spaces of type A.

```
A z;
z:=t;
for j:=1 until i-1 by 1 do
     if z::B          /* if z is of type B*/
       then z:=c of z;
       else go to error;
     fi; od;
if z::integer
  then z;
  else b of z;
fi;
```

Applied to the variable t in figure 3.1.1 it will yield:
for i≤1 3; for i=2 2; for i=3 1; for i>3 error

Figure 3.1.2 Retrieving the ith value of a sequence of type A.

Figure 3.1.1 shows an example where values are created using type A as basis. This creation does not cause any ambiguity, since every component data space is well known when used to compose a further instance of a type A data space. Observe that, theoretically at least, this form is used in GEDANKEN[Rey2] for the definition of vectors.

The information contained in the data spaces, created as shown in figure 3.1.1, can also be retrieved. In figure

3.1.2 we show an "ALGOL68" portion of coding exemplifying how to gain access to the i'th element of a sequence using type A as description.

Defn. 3.1.6 An acyclic composite type is a type that, for all subtypes, none of these subtypes contains itself as a subtype.

From what has been mentioned so far, it follows that acyclic composite types play an important role whenever spaces are allocated a priori, and when this allocation is based on the type definition of this space. The union type example above shows one case of cyclic composite types. This union type can be implemented in such a way, that there are no explicit name values. Therefore, even when using the convention 3.1.5, we are unable to break down the cycle of type A. From this we may conclude:

Fact 3.1.7 The restriction of types to only acyclic composite types implies a restriction of the power of expression.

We will not discuss here the merits of allowing cyclic composite types. Nor will we examine whether allowing such types is required in practice, and/or whether it substancially increases the probability to make subtle mistakes. We will only mention that, if we allow types to be functionally defined, it is impossible, in the general case, to decide whether the type is cyclic or not.

3.2 Type Operations on Spaces.

We will call typing the operation of associating a type with a space. Typing can be performed statically or dynamically. Static space allocation does not imply static typing. For instance, two different types may be associated with one single space by means of an EQUIVALENCE in FORTRAN IV. Furthermore, the ANSI standard[ANS1] requires that previous to the use of any of such variables, this variable must have been defined [i.e. assigned] with the appropriate type. Thus, although the implementations seldomly perform this checking, FORTRAN IV EQUIVALENCE establishes dynamic typing for statically allocated spaces.

Up to now we have assumed the typing operation to be instantaneous. However, this is not necessarily true. It is not even necessary that typing is an indivisible operation. Thus, there is a transient state, during which a data space being typed is neither of the old type, nor of the new type. It is clear that any access to the data space during this being typed interval is a valid access, as long as it is performed by the typing operation itself. However, if some other program section tries to access this data space during this interval, this other access must perfectly understand the typing operation. This follows from the fact, that the information content of the data space in question may be meaningless.

We will call T_i typing interval of a data space α, the time interval starting at the instant the type T_i is associated with α, and ending immediately preceding the instant a new typing operation is performed on α, or, then, immediately preceding the deallocation of α. It follows immediately that any typing operation initiates a new

typing interval, regardless of whether the type of the data space has effectively been modified. We will assume that whenever a space is being allocated, it is automatically typed with the type no type, i.e. undefined type. This type means that there is a storage space, but that its contents have no meaning. Observe the difference between a no type data space, and an undefined value. The latter has a given type, but stands for a data space which contents have not been initialized, i.e. defined after a new typing interval has been established. For statically typed spaces we will assume that the typing operation is instantaneous, i.e. timeless. Furthermore, we will assume that it occurs immediately after the allocation of the space having been completed. The net effect is then the same as if the space had been allocated with that particular type, without passing through the stage of being associated with no type.

We may conclude from the preceding paragraph, that dynamic space allocation implies dynamic typing of this space. It does not imply, however, that the types of the spaces accessible through the same locator function are necessarily different. Thus dynamic space allocation does not imply dynamic type checking. Nor does it imply dynamic type association with textual names, i.e. locator functions.

Defn. 3.2.1 A data space may be:

a- sequentially multityped if, at each instant, there is exactly one type associated with the data space;

b- parallel multityped if there are two or more types associated with the data space at some given instant.

Explicit typing does not imply sequentially multityped spaces. For example, it is possible to associate several types with one space in FORTRAN IV by means of any of COMMON, EQUIVALENCE or parameter association. Implicit typing through the use of operators, e.g. BLISS, B, BCPL, does not imply effectively parallel multityped spaces. This follows from the freedom the programmer has, to use a set of operators, which consistently attribute the same type to the space being accessed. In both cases, the burden of assuring meaningful use of data spaces is left to the user.

Consider now an ALGOL68 data space of type (mode) union. Such a data space has only one type during its dynamic scope. Its subspaces though, may have several different types during execution. Similarly the variant part of a type (record) in PASCAL is frozen when a given value of this type has been created. However, this does not mean that the data space containing this value is also frozen with regard to the possible variants it could hold.

Whenever a data space is accessed, the contents of this data space are implicitly typed by the access path. This occurs in any language and machine. Of course, in languages such as SNOBOL4 or GEDANKEN, data spaces are always of type "universal union". Thus, for this kind of languages, the following discussion becomes uninteresting. Now, if the machine performs some sort of type checking, e.g. the Burroughs B6700, the access paths implicitly type the data spaces being accessed with a given, hardware defined, union type.

On the other hand, the data space itself contains data of some type. That is, the data contained within a data space has a definite meaning within the program. For

example, a "prime integer" typed data space, contains not just some integer value, but contains a prime number encoded as an integer. Notice that the type of the space is a property of the information contained within that space, i.e. a property of the space itself.

There are then two types associated with each access. One is the access path implied type τ_a. The other is the data space implied type τ_s. An access to a data space α is said to be <u>type-wise correct</u>, if τ_a and τ_s are such that α is sequentially multityped. In sections 3.4 and 3.5 we will study the conditions which must be satisfied in order to assure type-wise correct accesses.

It should be clear that, whenever a program is written, we do want the data spaces to be sequentially multityped. This follows from our desire to obtain meaningful results. That is, at each instant we keep track of what the contents of a given data space are. If the language processor enforces type-wise correct access, we will be assured mechanically that, at least from the point of view of accesses, our program is correct. On the other hand, as we shall see, this implies a lot of writing effort by the programmer. Many languages which enforce type-wise correct accesses may become obnoxious when one attempts to do something which had not been forseen by the designers. For example, in PASCAL there are scalar types and variant records. These types are extremely well suited for business data processing. However, it is a common problem in business data processing that input data is incorrect. Thus, in PASCAL we must read field by field and perform all required checks before any record can be built, even if the input record is correct, as the majority tend to be. What we observe here, is that a definite duplication of efforts

exists. This duplication of efforts could have been avoided by allowing to define error handling interrupts. That is, we would allow the programmer to produce a detailed error checking procedure and which is invoked only if an error occured.

Defn. 3.2.2 Let τ be a type. The type set T of τ, is the set of all values of type τ. The type space TS of τ, is the set of all bit configurations which could be held by a data space, or encoding, of this type.

Let τ_0 be a FORTRAN IV LOGICAL type. By definition, the type set is {.TRUE.,.FALSE.}. The type space is machine dependent though. This follows from the ANSI requirement that a LOGICAL typed space occupies one space unit, e.g. word. Thus in an IBM/360, the type space TS_0 of τ_0 contains 2**32 elements.

Let τ_1 be a (red,yellow,green) PASCAL scalar type. The type set is clearly {red,yellow,green}. The type space is again implementation dependent. In a compact implementation, TS_1 would usually contain 4 different elements, since T_1 can be encoded using 2 bits.

Let τ_2 be a linked binary tree type. The type set is theoretically infinite. Cbserve that values of type binary tree do not take the implementation into account. Thus, the existence of pointers is ignored from the point of view of a "binary tree" value. The type space, however, will consider pointer fields, if any, as if they could hold any value what so ever, regardless of any inconsistency which might occur in doing so.

Let τ_3 and τ_4 be two finite scalar types. The elements in the cartesian product T_3xT_4 could be encoded in some form. Thus we would have a new type, say τ_5. The type set T_5 is T_3xT_4 by construction. A possible form to implement such a type is by multiplication, addition and extraction by means of modulos[Knu1,Hoa5]. What can be noticed in such an encoding is that the type space of either τ_3 or τ_4, is no longer an integer power of two. Thus, the subtypes of τ_5 no longer possess clear cut bit boundaries. By definition however, the type spaces TS_3 and TS_4 will be the set of values which could be extracted from a value of type τ_5, under the supposition of being of type τ_3 and τ_4 respectively.

Defn. 3.2.3 Let T_1 and T_2 be two type sets. A conversion $C_{12}:T_1->T_2$, is a (partial) function which takes elements of T_1 into elements of T_2.

With respect to conversions, we may classify type sets as follows:

a- T_1 is C_{12} convertible to T_2, if there exist two elements, t_1 in T_1 and t_2 in T_2, such that $C_{12}(t_1)=t_2$;

b- T_1 is always C_{12} convertible to T_2, if T_1 is C_{12} convertible, and for any value t_1 in T_1, $C_{12}(t_1)$ is in T_2;

c- T_1 C_{12} completely converts to T_2, if T_1 is C_{12} convertible to T_2 and C_{12} is onto, i.e. for each t_2 in T_2, there is a t_1 in T_1 such that $C_{12}(t_1)=t_2$.

Observe that conversion does not necessarily maintain the "meaning" of the converted value. For instance, converting real to integer corresponds also to some form of

"rounding". Observe furthermore, that coercion, as defined in ALGOL68, is not the same as conversion. Coercion in ALGOL68 stands for several rules according to which a conversion, or sequence of conversions, is chosen in order to convert a given type (a priori mode) to the required type (a posteriori mode). We will not study coercion rules in this dissertation.

Conversions may be both language and machine dependent. For example, the effect of converting an integer into a bit string may depend on the word size, as well as on the language being used. For instance, when converting to bit strings in PL/1, the conversion is based on the leftmost portion of the bit string. That is, if the target string is shorter, the object string will be truncated on the right, and conversely, if it is longer, the object string will be padded with zero bits on the right.

There may be several conversion functions defined for a group of type sets. For example, a conversion from real to integer could be any of: truncation; rounding; ceiling; floor etc. Finally, if users are allowed to define their own types, they should also be enabled to define conversion functions having these types as domains and/or ranges.

Defn. 3.2.4 A conversion $C_{12}:T_1->T_2$ is said to be an identity conversion, if, for all t_1 in T_1 and t_2 in T_2 such that $C_{12}(t_1)=t_2$:

i- the bit configuration of t_1 is not affected by C_{12}; and

ii- the bit configurations of t_1 and t_2 are equal; and

iii- T_1 is completely C_{12} convertible to T_2.

C_{12} is said to be a <u>partial identity conversion</u>, if all conditions, except complete C_{12} conversion, hold.

Notice that not all pairs of type sets admit an identity conversion. For instance, conversion from double precision floating point to a single precision floating point type, cannot be achieved by means of an identity conversion. Depending on the hardware, though, a two element single precision floating point type may be identically converted to a double precision floating point type.

The PL/1 function or pseudo variable UNSPEC may be regarded as a restricted form of an identity conversion. It is restricted since the domain must be a bit string, when used as a pseudo variable, or the range is a bit string, when used as a function. This may cause additional conversions to occur. These additional conversions are not necessarily identity conversions. Therefore, the results of using UNSPEC may cause unexpected errors to occur. Furthermore, UNSPEC is a universally defined function. In our case, identity conversions exist only within a limited scope. Also, we must define explicitly the range and domain data types to which a given identity conversion applies. This is expected to decrease both the ability to misuse the language, as well as the possibility of unexpected errors.

Let us consider now a hash index generating function. This function is a conversion from character string to a subset of the set of integers. In order to obtain the first hashing index, the recomended method[Lum1] is to consider the string as an integer, and obtain the remainder of the

division of this integer by the size of the table†. Now, if the identity conversion from string to integer were available, this hash conversion could easily be implemented. This motivates making some identity conversions available to the user. Of course, the price paid is that some machine or implementation dependency may be caused. On the other hand, it is impossible to foresee all possible conversion functions, mainly if the user may define his own types. The discussion in this paragraph may be summarized by:

Fact 3.2.5 The exclusion of identity conversions from some programming language, restricts the power of expression of that language.

Lemma 3.2.6 Parallel multityped spaces may be simulated by means of an identity conversion, and vice versa.

Proof. It suffices to notice that a parallel multityped space necessarily contains exactly one bit pattern, regardless of which type is actually being implied. That is, a parallel multityped space represents a trivial identity conversion among the types which could be associated with the multityped data space.

From this we may conclude that restricting languages to multitype spaces only sequentially, does not curtail the power of expression of the language, as long as identity conversions exist. From now on we will then assume, that data spaces are always sequentially multityped, except when explicitly said to the contrary.

We will call *type checking* the operation of determining the current type, or types, associated with a

---

† We are not interested in how to solve possible collisions here.

data space. Observe that, for statically typed data spaces, this operation plays a major role at compile time. Notice that static typing does not imply static type checking. For example, in ALGOL60 the actual and formal parameter association does not require the same type for both parameters. Thus, when accessing formal name parameters, type checking must be performed†.

One of the interesting aspects of sequentially multityped data spaces, is that they usually ease type checking. Furthermore, we only have to make sure at each instant, that the value contained in the data space is one of the elements in the type set, without having to verify this for several types. Furthermore, due to the mutual exclusive typing of sequentially multityped data spaces, we may enforce space disjointness for intrinsically identical types, by means of null information types. Finally, several operators allow values of several types as input parameters [e.g. the operator "+"]. Such operators may also behave differently, depending on the input types. Enforcing sequential multityped spaces, automatically prevents any such ambiguities.

We will call conversion predicate P_{12} of the conversion C_{12}, a predicate which returns true, iff for a given value t_1 in T_1, there is a value t_2 in T_2, such that $C_{12}(t_1)=t_2$. Following are the reasons why such a predicate may fail:

a- C_{12} does not apply to T_1 and/or T_2, i.e. incorrectly chosen conversion function.

b- t_1 is not in the domain of C_{12}. For example, when

† Note that some implementations build multiple entry point thunks[Ing1] in order to avoid an explicit type checking.

converting character strings to integers, a value such as '1A2' cannot be converted.

c- $C_{12}(t_1)$ is not in T_2. That is, C_{12} properly "computes" a t_2 "value", but this "value" does not belong to T_2. For example, when attempting to convert the string '1234' to integer, the conversion will fail if this "integer" type is implemented with less than 11 bits.

With respect to types, we may classify access to data spaces in the following way:

a- converted access form, whenever a data space is accessed, a type check is performed. If the type found is different from the expected type, a conversion will be attempted, or an error will be reported.

b- unconverted access form, no type check is performed when accessing the data space.

It is immediate that unconverted access form does not cost any type checking overhead at run time. However, it leaves to the programmer the burden of certifying access correctness. This form of access is common to most assembly languages and also to languages such as B, BCPL and BLISS. Observe that a typed language does not imply converted access. This follows from our more general interpretation of the concept "type". Thus, a binary tree is a type, regardless of how it has been implemented. However, if the language does not allow to define a type "binary tree", the programmer must certify that any access to a value of type binary tree (or a subvalue thereof) is a valid access. Observe also, that with respect to higher level structures, the problems to be resolved when attempting to certify the

use of such a structure are much greater. Furthermore, the probability of existence of subtle mistakes is considerably higher.

Converted access does not necessarily have to be expensive. This follows from the fact that it may be performed at compile time. Observe that this checking may not be possible right at the first compile step, but rather at one subsequent step. Thus, there may be a need for creating library procedures which are "typeless" in the sense that the complete type description be filled in at some later compilation step. For example, the required sizes of arrays may perhaps be unknown at the first compile step, but they are known previous to the completion of all compilations, e.g. at load time.

Parallel multityped spaces do not imply unconverted access forms. Consider for example the following two FORTRAN IV statements:

```
COMMON /A/ IX     in SUB1
COMMON /A/ AX     in SUB2
```

within SUB1, IX will be considered as an integer, and thus, may cause conversions to be inserted into the code if used in a context where a different type, e.g. real, is required. A similar thing occurs also for AX in SUB2. However, through the COMMON space association mechanism, these two spaces are one and the same space, which is then parallel multityped, and accessed in a converted form.

In the current literature, conversion functions are frequently called <u>data transfer functions</u>[Sta1,Hoa5,Nau2]. We have chosen the term "conversion", for we feel that this is intuitively closer to the concept it describes. Furthermore, some special conversion functions are

frequently denoted by an individual name. The function which converts several types into one structured type, is called a constructor[Hoa5,Sta1]. Conversely, the conversion function which obtains a subtype value of some value, is called a selector.

When considering types in a more general form, any procedure could be thought of as being a conversion. This follows from the fact that the input parameters may be composed into one type, as also the output parameters may be. The boundary of which constitutes a procedure, and of which constitutes a conversion, is quite ill defined. For example, the hash index generator above may be thought as being either a function or a conversion, without stretching too much either of the concepts. We will say then, that a procedure is a conversion, only if it has explicitly been built for that purpose.

3.3 Type Descriptors as Values.

Type descriptors are information, and as such, they could be considered as being values of type type. This implies then, that we could have also variables of type type. Although not explicitly, Hoare describes some operations on values of type type[Hoa5], e.g. concatenation [cartesian product] and union. These operations are usually performed at compile time. Once the program has been compiled, the type descriptors become constant. In some languages these constants are of no further interest and, consequently, may be discarded, e.g. FORTRAN IV with respect to non-dimensioned types. In other languages, these type constants are required at run time and must be kept during execution. For example, the type descriptors for all variables in the heap

must be kept in ALGOL68, otherwise the garbage collector could fail.

In SNOBOL4 users may define their own data types at run time. Thus there are "type variables" in SNOBOL4. In fact, only nodes and their field names are defined, since data spaces themselves define the type of the data they contain. Furthermore, in SNOBOL4 the operation of assigning a type value to a variable is, a special purpose function [DEFINE]. There is thus an explicit difference between "normal" assignments and type value assignment.

The classic operations on type values are:

a- concatenation, i.e. structuring data types. Examples of type concatenation are: COBOL and PL/1 structures; struct in ALGOL68; records in ALGOL-W and PASCAL; classes in SIMULA67; blocks in any block structured language. In all these cases the types being concatenated may be heterogeneous. When the types being concatenated are homogeneous, the resulting type is usually called an array. Notice that we do not expect that concatenated data types be necessarily implemented in a contiguous space. For example, a SNOBOL4 user defined data type is a concatenated data type. However, each field refers to a space of its own, which is usually non-contiguous to the space of the concatenated type.

b- union, i.e. the effective type is any of a, usually small, set of type values. Examples are union in ALGOL68 and variant records in PASCAL. In SNOBOL4 and GEDANKEN, a data space defines the type of the value it contains. Thus, in both SNOBOL4 and GEDANKEN, all data spaces are of type union over the set of all

possible types. Such union types will be called universal union types. In a sequentially multityped environment, whenever a union typed data space is accessed, we must first establish the effective type of that space. This is usually accomplished by run time checking. It could, however, be accomplished directly by the programmer, e.g. when programming in B or BLISS.

Although we have borrowed the nomenclature from ALGOL68, we do not expect the set of effective types of some union type to be defined statically. Observe that there are no restrictions with respect to dynamically defined types and union types. Even under these conditions, sequential multityping is mechanically enforceable by keeping an actual effective type descriptor field within the union typed space. The value of this field describes the type of the information currently carried by the union typed space. In L^6[Kno2] node descriptors are defined at run time. Nodes may be accessed at any given time using any of the currently known node descriptors. Observe, though, that L^6 does not enforce sequential multityping, since nodes may overlapp and no control is made with respect to the current meaning of the node being accessed.

Union types could be defined implicitly by the language processor. For instance, in SNOBOL4 all data spaces are of type "universal union". The existence of union types could even be hidden from the user. For instance, in ALTRAN[Hal1,Bro3,Bro4] there are conversions defined between the types "integer", "rational", "algebraic" and "real". Promotion is a conversion operation from a type τ_1 to a type τ_2, where τ_1 is a (logical) subset of τ_2, e.g. "integer" to "rational" conversion in ALTRAN.

Demotion is the inverse conversion of a promotion. Observe that promotions are always possible, whereas demotions are possible only in some cases, e.g. demoting the rational 1/2 to "integer" is not possible. If there is a definite gain in storage economy and/or in execution time economy, the implementation may opt to store values always in the "most demoted" form. The user still declares a unique type for a given textual name and has the impression that this is also the type with which the program will effectively operate, in fact a union type is being used though.

In chapter 4 we will see that, in order to assure the sequentially multityped environment, pointers must imply the type of the data space they refer to. Thus, borrowing ALGOL68 notation, when defining a "ref to type" type, we are not performing an operation on values of type type. Rather, we are just restricting the set of types which a name typed value may characterize.

Although union types are usually implemented as a structure $\langle\langle \tau_a, type\rangle, \langle \alpha, \tau_a\rangle\rangle$ where τ_a is the actual effective type of the subspace α, this is not always the case. Consider, for instance, equivalenced variables in FORTRAN IV. If the rules stated by ANSI are obeyed, such a space must be considered as a sequentially multityped (union) space. The actual effective type is provided implicitly by the programmer. Observe that when we allow union types where the actual effective type descriptor is implied, we are in fact allowing the existence of parallel multityped data spaces. This follows from the fact, that the locator function accessing such a space has no means to certify the type-wise correctness of the access in this case.

A data space α is said to be object typed, if the data space itself defines, or refines the definition of the type of the data it contains. A union typed data space where the the actual effective type is explicitly provided, is an example of an object typed data space.

Object typed data spaces are frequently called selfdefining data spaces. Observe, though, that an object typed data space consists of at least two subspaces. One of these subspaces contains the type descriptor, or a reference (encoding) to it. The other subspace contains the actual data. Thus, usually, an object typed space identifies the contents of one of its subspaces, but not of itself.

We say that a subspace β of a data space α is positionally typed, if the relative position of β within α implies the type of β. For example, the parity bits of a core position are positionally typed.

Suppose that α is object typed. There must then be an actual effective type descriptor field. Suppose that this descriptor field is not positionally typed. There must then be a way to identify what is the actual effective type descriptor, since only then we are able to tell what the actual data is. Now this is possible only if the encoding of the type descriptor is such, that it could not possibly be confused with any of the bit patterns of the actual data space. This is so, since otherwise there would be no mechanical way to identify the type descriptor. Such a form of identifying the actual effective type descriptor is possible only if the descriptor's value (bit configuration) is different from the bit configurations of all possible actual data values. Suppose now the type descriptor were

positionally typed instead. We observe immediately that there would be no major difficulty in finding the actual effective type. Notice that the "type descriptor bits" of a Burroughs B6700 are positionally typed. We have to stress the fact, that positional typing depends on the medium and the address of the subspace. It does not depend, however, on the extent of this subspace. That is, the fields do not necessarily have to be of a fixed size, as long as the starting (or ending) address of the field implies the type of the field.

Formalizing we have then:

Theorem 3.3.1 Object typing of a data space is possible iff:

a- the type descriptor or its encoding is positionally typed; or

b- the type descriptor, or its encoding, is identifiable by restricting the type set encoding of the actual data subspace.

In general object typing increases the run time cost, since type checking has to be performed at run time. It has several advantages though. A textual name may represent several data spaces, each of a given type. The selection of the actual data space to be accessed is then based on the type expected. For example, in SNOBOL4 a textual name may represent a label, a function and/or an actual data element. The choice is based on the syntactical occurence of this textual name, i.e. function call, go to field, or the "applicative" part of the statement. Another advantage of object typing is when some subtype admits several forms of storage. For example, in ALTRAN, the actual value of a potentially algebraic value could be just numeric. Storing

it as such, instead of as an algebraic value, overwhelms the cost of performing a run time type check.

Lemma 3.3.2 In a sequentially multityped environment, the type of any contiguous data space α is obtained by the type operations of concatenation and union.

Proof. That the operations of type concatenation and type union are capable of defining contiguous spaces is a well known fact, c.f. Hoare[Hoa5].

Suppose now that there is some operation which is neither concatenation nor union, and which is required in order to describe a given contiguous data space. This operation must associate more than one type, since otherwise we would have a degenerate case of concatenation. Let τ_i and τ_j be two of the type values used by this operation. The corresponding subspaces are α_i and α_j. They cannot possibly be one and the same space, since then we would fall into the case of type union. They can also not be totally disjoint, since, by α being contiguous, we would have a space $\alpha_i\alpha_k\alpha_j$ of type $\tau_i\tau_k\tau_j$, which is a type concatenation. Thus, α_i and α_j must be different and also intercept, say in α_{ij}. Since we are considering a sequentially multityped environment, τ_i and τ_j each must posses a subtype τ_{ij} and τ_{ji} such that $\tau_{ij}=\tau_{ji}$, and where τ_{ij} is the type of α_{ij}. But then we can break α_i and α_j into three subspaces $\alpha_i'\alpha_{ij}\alpha_j'$ which are of type $\tau_i'\tau_{ij}\tau_j'$, that is, a concatenation of types, contradiction.

In accordance with Hoare[Hoa5], type descriptors are not only responsible to define possible storage layouts,

but they also describe the operations which could be performed on values of the given type. Notice that this also corresponds to the concept of data structures. This is precisely what we are aiming at with our generalized view of types. Usually the storage layout for values of a given type is defined by the semantics and/or the implementation of a language.

The basic operations which must be defined for each type are fetch, store and access [i.e. fetch and/or store]. Usually the programmer needs not to define these operations, since they are implied by the language semantics which apply to concatenation and union. Notice though, that in ALGOL68 we have the power of defining the existence or non existence of the store and access operations. E.g.

```
real pi=3.1415;
```

defines pi having just the fetch operation defined. Whereas:

```
real pi:=3.1415;
```

allows any operation to be performed on pi.

A further reason to allow the programmer to define his own access requirements, is that we enable him to protect information, or to disguise the existence of additional information. For example, if we use the buddy system[Kno1, Knu1], or the Fibonacci system[Hir1] for that matter, to perform dynamic memory allocation, we must keep additional information defining the companion data space. Of course, a user making use of spaces provided by such a dynamic storage allocation mechanism, should not be aware of the existence of this additional information. Similarly for union typed spaces we must keep information describing the effective type of the data in this space. This additional

information may be accessed, as in the case of conformity relations in ALGOL68. It may even be changed in a controlled environment, as it is when retyping the "selfdescribing" subspace of a union typed data space in ALGOL68.

The protection could also be, that certain access is valid only within a given context, e.g. a specific set of procedures or users. The advantage in doing so lies not only in the capability to restrict the dissemination of information, but also in the capability to protect information against the the contamination by incorrect program sections. Finally, allowing only a selective access will decrease module coupling as defined by Myers[Mye1], increasing thus the "modularibility" of a program.

Although intuitively quite similar, there is a definite difference between access typed values and ref typed values. An access typed value is a name value in the same way as a ref typed value is. However, access typed values are never values on their own right. That means, if an access typed textual name occurs at the left hand side of an assignment, it will always deliver a reference to the data space it currently is bound to. In ALGOL68 terminology, an access typed value is, thus, always dereferenced. Usually access typed values define an access path rather than a simple name typed value. Access typed values still satisfy convention 3.1.5. However, the external appearance is as if the data space they characterize would stand in their place.

Besides the access operations described above, a user may also define manipulative operations. Such operations are particularly interesting when a certain type is

designed for a specific kind of information. For example, when defining square arrays for the purpose of holding the coefficients of a set of linear equations, it is worthwhile to define also operations, such as matrix addition, multiplication and inversion.

Notice though, that if the access operations are available, we need not provide extra manipulative operations. This follows from the fact that we must know what we are accessing, and, consequently, we could use a set of standard operations to perform the required operation. For example, using selection and real addition, we are able to define addition of arrays, provided that the access operation is defined for this array. Formalizing we have:

Lemma 3.3.3 Let τ be a type. If the access operations are defined for τ and for all its subtypes, then any manipulative operation can be implemented for values of type τ.

In some circumstances, we may want to protect data spaces in such a way, that only manipulative operations are made available to the outer scope of the type definition. In section 3.4 we will discuss how to define precisely the scope of names with respect to modules, or program sections.

Again we are faced with the problem that we cannot foresee all possible operations that might be required. There is no major difficulty if we satisfy the conditions of lemma 3.3.3 though. However, we may not wish to do so, since we might want to protect information against uncontrolled access. To force a user to define a complete new descriptor, seems not to be a good choice. This shows that we should allow type descriptors to be extensible. The

problem is now how to solve the conflicts between extension and protection. Again we resort to allowing specific information to be accessed only within a specified scope. Thus, we have now protection sets, not necessarily disjoint, describing which information may be accessed under which special condition.

Besides protection mechanisms, we also must provide operators which enable a type descriptor to be extended or pruned. Furthermore, we must be able to link these extensions into the text in the appropriate places. In figure 3.3.4 we show an example of a type descriptor extension. In order to understand it, the following discussion on incomplete type descriptors should be read first.

By means of predicates defined by the type descriptors, we may assure that values of a given type satisfy some given condition. This is particularly interesting when the type set maps onto non-contiguous elements in the type space. For instance, a type "prime number" could be defined. Whenever a value of this type is assigned, a check verifying whether the number is really prime should be performed. Observe that, by means of extensions, some type definitions can be created from other type definitions, by defining or redefining the conditions which must hold for values of these new types.

The last consideration is that of human engineering. We must be able to measure the amount of "annoyance" a programmer is willing to accept, in order to obtain increased performance and a higher degree of mechanical certification of his programs. The answers to these questions are not at all clear. First, because they depend on each individuals taste. Second, because it is not clear

that, for a more involved program, a substancial amount of mechanical verification is possible, apart from syntax checking. We will not pursue further these questions, since they fall out of the scope of this dissertation.

From the preceding paragraph we may conclude, that there should be a set of system "defaults" which would reduce the amount of effort spent by the programmer. For example, if no operation is mentioned, then the usual access methods apply.

When producing a major piece of software, it may be worthwhile to individually certify each of the composing modules. Since normally these modules will operate in a well defined environment, we would like at least to be able to simulate this environment when testing a given module. This suggests that there be a set of interchangeable type definitions, in order that the one closest to the needs at the given moment could be chosen. We understand as "being interchangeable", a type descriptor which allows the same set of operations on the same input parameters. The individual behaviour is irrelevant, as long as the results produced are not affected by which of the definitions is effectively been used for a given run.

Similar problems may also occur, when a system is defined for a large set of input values. For instance, the operations defined for arrays may vary largely with respect to properties such as sparseness, symmetry etc.

In the case of extension of type descriptors, there is again a conflict. This is usually a consequence of the fact that extension is a type descriptor dependent operation. It follows then, that a given set of type descriptors is only apparently interchangeable. It is really interchangeable if

they are also interchangeable with respect to all extensions. This may be achieved either by assuring that the extensions are independent of the particular descriptor of the interchangeable set. Or it is possible, by creating several extensions, one for each type descriptor, and assuring that the current extension always applies to the actual descriptor. This can be checked quite easily by associating a unique identification, say creation stamp, e.g. creation time or global counter value, to the type descriptor being extended. When an extension is being defined, the underlying type descriptor must be known, thus its creation stamp can be copied. On a later occasion, when the type descriptor and the extension are combined, the extension is valid iff its creation stamp matches that of the extended type descriptor.

When defining certain types, we may not know, or do not want to know, the exact type description of all composing subtypes. For example, the access methods of an array are usually the same, regardless of whether it is an array of integer or real values. This suggests then to allow the programmer to define types in an incomplete fashion. The information may be completed at some later textual point, or a later compile step, or, finally, at run time. Using again arrays as example, the number of rows and columns needed for a specific application is frequently not known during the first compile step. However, when producing the final assembly of the system, still within compile time, these values could be filled in. A similar argument holds also for constants. Some constants may be known for a given application, but not when writing the program, and could be filled at some later compile step. Now, if we observe that type definitions could be viewed as

constants, there is no major reason, except for complexity, which would prevent us from defining types in successive steps.

```
type stack of (type user_type)=
   begin stack;
      type node=struct(ref node next; user_type info);
      ref node first:=null;
      outside_scope operations;
         user_type access function top=first->info;
         function push(user_type value)=
            first:=new node(next::first, info::value);
         logical function pop=
            if first≠null
               then begin pop;
                        first:=first->next;
                        true;
                    end pop;
               else false;
            fi;
      end operations;
   end stack;
```

Figure 3.3.1 List implementation of the type "stack of user_type".

We will call type macro, a macro which, upon expansion, delivers a completed type description, or another type macro. The discussion in preceding paragraph shows a possible use of type macros.

In figure 3.3.1 we show the definition of an incomplete type "stack of user_type". From the point of view of the stack storage mechanism, it is completely

```
type stack of (type user_type) =
   begin stack;
      user_type space[50];          /* stack area */
      integer first:=0;
      outside_scope operations;
         user_type access function top=space[first];
         function push(user_type info) =
                  space[first:=first+1]:=info;
         logical function pop=
            if first>0
               then begin;
                  first:=first-1;
                  true;
               end;
               else false;
             fi;
      end operations;
   end stack;
```

Figure 3.3.2 Array implementation of the type "stack of user_type".

```
type x=struct(integer a, real b);
stack of(x) saver1;
stack of(integer) integer_stack;
saver1.push( );  /* if no parameter then send undefined */
with s1=saver1.top do s1.a:=0; s1.b:=1.; od;
integer_stack.push(1);
i:=i+integer_stack.top;
```

Figure 3.3.3 Example of the use of the type "stack of user_type".

irrelevant what the effective type of the data being stored is. Having this in mind, we may define all the operations applying to a stack, using the type of the information to be stacked as a parameter.

In figure 3.3.2 we show the array implementation of the type "stack of user_type". Observe that both types in figures 3.3.1 and 3.3.2 are "identical" from the user's point of view. The major difference is that the array implementaion defines an upper bound on the stack size. In both examples we assume that access functions always return null if their evaluation fails. Thus a null pointer, or an index out of bounds, will return a null access typed value.

In figure 3.3.3 we show some statements which use either of the type descriptors "stack of user_type" defined in figures 3.3.1 and 3.3.2. When defining the type of the variables saver1 and integer_stack, we effectively complete the information of the type "stack of user_type". Observe that "user_type" is a parameter of type type, and that this parameter is filled at compile time.

There are several ways to implement such a set of constructs. The simplest one would be to generate an instance of each of the functions, for each of the ways the type is completed. Observe also, that the type "stack of user_type" (fig 3.3.1) defines the local variables "first" and "user_type". Each textual name implying a space of type "stack of user_type", e.g. saver1 and integer_stack, will refer also to the data space containing these local variables. Furthermore, for each of the textual names, there is an instance of "first". However, this data space is accessible only by the functions defined by the type "stack of user_type". Observe also that, in the present

case, the local value "user_type" can be discarded after compilation.

Observe the similarity of this implementation and that used for classes in SIMULA67. Observe also, that in SIMULA67 all local variables are accessible, whereas in our case this is not possible, e.g. the variable "first".

Another implementation would be to use only one set of procedures. In this case, each of the procedures would receive also the type as a parameter. This parameter would have to be included in an implicit fashion, since the user should be unaware of the implementation. Observe that this implemetation is natural to SNOBOL4 and GEDANKEN, since in these languages, data spaces define their own type.

Which of the two implementations to chose is quite difficult to be done mechanically. This follows from the fact that, in order to correctly decide, we must take into consideration the cost implied by a given implementation.

In figure 3.3.4 we show an example of a type descriptor extension. This extension is sensible to the underlying type descriptor. It has been designed to extend the list implementation of the type "stack of user_type" (fig 3.3.1). Each identifier created by the extension must be appropriately placed in the text of the descriptor being extended. This is achieved by the in <place> constructs, where <place> is any of the textual scope defining constructs, e.g. begin, scope etc. The reason for the existence of a boolean function user_type.equal, is that we are unable to tell how equality of values of type "user_type" is defined.

```
type symbol_stack =
extend stack with
   begin extension;
      ref node pos in local;
      user_type access function search(user_type name) =
         begin search;   /* find node bearing "equal" data */
            pos:=first;
            while pos≠null do
               if user_type.equal(name,pos->info)
                  then begin found;
                        pos->info;
                        exit search;
                     end found;
                  else pos:=pos->next;
               fi;
            od;
            null;
         end search; in outside_scope;
   end extension;
```

Figure 3.3.4 Example of an extension of a type descriptor.

In the same way as we defined "stack of user_type", we could also have defined "trees of user_type", "queues of user_type", "arrays of user_type" etc. Observe the richness of types we gain in doing so, and also the adaptivity of types to a given program's needs. Observe furthermore, that incompletely defining types allows us to certify once and for all, all occurrences of such a type, regardless of its actual parameters. That is, we are effectively increasing the structuring of the program. Finally, the annoyance of having to write more when defining such a type, is greatly offset by the reduction of writing when such a type is used

frequently in a given program. It is offset even more, if the language translator is capable of reading type definitions from external files, since, in this case, such a type descriptor could be defined once only for a set of programs.

To expect a language to provide a great richness of types is, to say the least, unrealistic. First, because the language would become topheavy. Not only this, it is also near to impossible to foresee all the types which could possibly be used by all programmers programming in that language. Second, language implementations tend to freeze the implementation of standard types. This standard implementation is not necessarily the best one for all cases. Finally, the cost of implementing the capability of incompletely defining types seems to be reasonably small.

Another interesting point with respect to type macros, is the ability to group data spaces on the medium. That is, by maintaining several free lists, we are able to keep data spaces which will be accessed in a close succession in such a way that the access cost is reduced. This is particularly important when considering backing store, or programs which execute in virtual memory machines.

Let τ be the type descriptor provided as an expansion parameter to a type macro. If τ is known to define spaces of bounded length only, space may be allocated by the type descriptor functions, even if the actual value is undefined. A consequence of this, is that the data of type τ may be processed and kept "locally" to the type descriptor. This reduces thus the amount of information traffic due to copying information. An example of the use of this property is shown by the construct:

```
saver1.push( );
```

in figure 3.3.3, where the value to be pushed is undefined, but its storage requirement is well known.

Suppose now that τ defines data contained in data spaces of unbounded length, e.g. lists. If the information of type τ is to be made internal to the type macro, first the bound has to be computed. Usually this is possible only by generating the value of type τ. This may increase considerably the amount of information traffic due to copying information. A pointer [reference] to a data space of type τ is obviously bounded in length. Thus if τ is a "pointer to type" type, the problem reduces to the first of the presented ones. However, the user is now responsible to make sure that dynamic storage is being handled properly.

The preceding discussion can be formalized by:

Lemma 3.3.4 A type macro may make internal spaces available containing yet to be defined values, iff the type parameters define bounded spaces, or the space requirements of a particular value are computable during execution.

As seen previously, a type macro may define its own local variables. The collection of all local spaces can obviously be considered as a concatenation of types. Furthermore, these local descriptors are usually bounded in length. Observe then, that we could define a type "stack of stack of user_type" as is shown in figure 3.3.5. Observe that this does not contradict lemma 3.3.4, since the variable length of the parameter type, is kept in dynamic storage, i.e. not internal to the type macro. Observe furthermore, that stack1 is a complete type descriptor, i.e. there is no further information needed for this type

descriptor. In figure 3.3.6 we show an example, where the recurrence is performed with still incomplete type descriptors.

```
type stack1 = stack of(integer);
```

this defines just a new type. No variable is being defined, thus also no space is reserved.

```
stack of(stack1) stack_of_stacks;
```

this defines a variable consisting of a stack of stacks. Since "stack of user_type" defines a local variable, this definition causes space to be allocated. However, only space for "first" in stack of stacks is allocated. Due to the static typing of stack1's user_type, the value of "user_type" is not needed at run time.

```
stack_of_stacks.push( );
```

this is a valid operation. It will create just a new stack frame of type stack1. There is no need for a parameter, since creation of values of type stack1 precinds any parameter. There is, however, space allocation involved with this operation. This space will contain the variable "first" of the new stack defined.

```
with ss=stack_of_stacks.top do ss.push(1); od;
```

this is also a valid operation. It will cause a new stack frame of type integer to be allocated to the topmost stack in the stack of stacks. There is no need to prefix "push(1)" with a stack identifier, since this is automatically done by means of the "with" construct. We could have written also:

```
stack_of_stacks.top.push(1)
```

which would achieve the same result.

Figure 3.3.5 Example of the recursive use of an incomplete definition, yielding a complete descriptor.

Theorem 3.3.5 Let τ be an incomplete type descriptor at some compilation step S_i. Run time type checking for statically typed data spaces based on type τ is required in the general case, if there is no compilation step S_j during which the type descriptor τ is completed.

```
stack of(stack of(type user_type)) stack_of_stacks;
```

this definition is valid, however it yields an incomplete descriptor. There is no problem, though, in allocating space for the variable "first" in stack_of_stacks.

```
stack_of_stacks.push( )
```

this operation is not valid, since we do not know the type of the new stack being created.

```
stack_of_stacks.push(integer);
```

this operation is valid, since it correctly completes the type of the new stack being created. This shows also that, at run time, there are now two local spaces associated with each stack, one for the type definition of the stack, the other for the variable "first"

Figure 3.3.6 Example of recursive use of incomplete type definitions yielding an incomplete type descriptor.

Proof. Suppose that τ has not been completed when starting execution. It is clear that there may be no allocated variable of type τ, since descriptive information to handle this variable would be missing. Therefore, previous to any typing operation, a completed instance of τ must be generated, say τ'. Since there could be several such instances, each one different from the other, we conclude that, in the general case, a type check is needed in order to determine the appropriate type of the data space being accessed■

A consequence of this theorem, is that incomplete type descriptors may act as union types if left open during execution. In principle this union type is defined over the set of all possible types. This may cause major difficulties both for the code generating procedure, as well as for the programmer using this feature. From this we may conclude, that the set of possible type parameters be

restricted and explicitly provided, e.g. by means of the union construct.

Incomplete type descriptors usually define polimorphic operators. That is, operators acting on values of type "user_type" are undefined until "user_type" is provided. Furthermore, for different "user_types" also different operations may effectively be performed, e.g. "+" of integers is different from "+" (concatenation) of "strings". Since the operations on values of type "user_type" must be defined within the "user_type" type descriptor, there is no major difficulty regarding the definition of the operations. The problems arise, though, when attempting to perform independent compilation, since frequently there will be insufficient information. We will come back to this problem in section 3.4.

3.4 Type Identification.

In section 3.2 we have seen that access paths always imply the type of the data spaces bound to them. Thus, when accessing α by means of a typed access path $\langle b,\tau_p \rangle$, α is implicitly typed τ_p. On the other hand, the type τ_s of the data space α, is a property of α itself. Therefore, if α is to be sequentially multityped, there must be an equality relation between the type τ_p implied by the access path, and the effective type τ_s of α.

Most of the following discussion is irrelevant for languages where data spaces are always "universal union" typed, e.g. SNOBOL4 and GEDANKEN. This is so, since in these languages data spaces always describe the type of their contents.

The usual definition of type equality, is based on the equality of storage layout, c.f. Ledgard[Led2], Morris[Mor2], Scheidig[Sch1], Lewis and Rosen[Lew1], Harrison[Har2]. This definition does not suit our concept of type. For example, let τ_p be the type "prime integers" and τ_i be the set of "integers". The storage layout is obviously the same. Using this definition of type equality, we would have to conclude then, that both types are equal. It is obvious, though, that the type sets are different and, consequently, the types cannot possibly be equal.

Another difficulty with the conventional concept of type equality, is that there is some meaning attached to a data space. This meaning is not necessarily reflected by its type descriptor. This may cause information of one type to be non-interchangeable with information of another type, although both types apparently define the same type set. For example, let α be a data space containing a students name, and let β be a data space containing a professors name. Borrowing PL/1 notation, let us suppose that both have been declared CHAR(40), thus their type space is equal. It is easy to verify now, that each of these data spaces may contain a value of a scalar set, which happens to be implemented as CHAR(40). For several reasons, this scalar set is not explicitly declared, thus one is led to assume that each element in the type space may represent a value of the given type. We would have to conclude then, again contradicting reality, that both types are equal. We will call this kind of type inequality a <u>semantic type inequality</u>.

Observe that in our first example, we could have defined a predicate which would test for membership of a given integer in the "prime integer" type set. This

predicate could be used then, as a functional encoding of this type set. In the second example, we cannot produce such a predicate, since such a predicate is not computable. We would have then to resort to enumeration of all elements in the type sets. It should be obvious that this is non satisfactory.

To determine whether two data spaces of the same type represent semantically different types is undecidable. Thus the user has to make the decision, but also the language must provide a facility to differentiate types which are apparently equal. Before showing how this could be achieved, we will study problems relative to identification and equality of types.

Defn. 3.4.1 Let τ_1 and τ_2 be two types. We say that τ_1 and τ_2 are theoretically equal iff:

a- their type sets are equal; and
b- their sets of valid operations are equal; and
c- their type spaces are equal; and
d- their subtypes are respectively theoretically equal.

Observe that the type space equality and the subtype type space equality imply implementation equality. We must take the implementation into consideration, since we are ultimately interested in information interchange, rather than in set theory.

In section 3.1 we saw following recursive type definition:

```
type A=union(integer;
             type B=struct(integer b; A c));
```

In order to allow finite computation, we must regard a type definition as a static entity, i.e. we do not follow subtype definitions which are based on recursion. Thus, it

satisfies item (d) of the theoretical type equality, if two types possess the same recurence graphs for all their subtypes.

The first problem with regard to this definition of type equality, is that type sets could be defined functionally, e.g. the "prime integers" set mentioned before. Such types impose severe restrictions, since it is undecidable, in the general case, whether two different functions compute the same results for the same input values. We must thus find an alternate way to identify data types. This identification should allow the definition of an "equality" relation which is at least as restrictive as the theoretical equality of types.

For each type descriptor there is a textual name. Some of these names are system defined, e.g. integer, real. Others are user defined, e.g. modes in ALGOL68. Not necessarily, though, must these textual names be known at compile time, e.g. SNOBOL4's DEFINE function defines type descriptor textual names at run time. Textual names may represent different things, depending of the textual reach within which they exist. Thus, a textual name could stand, among others, for several different type descriptors.

Defn. 3.4.2 An assignation interval of a space α, is a time interval starting at the instant α is write accessed, and ending at an instant immediately preceding deallocation of α, or the next write access to α.

In a deterministic environment, a textual name within a given reach can represent one only data space at each instant. This shows then, that following definition is sound, i.e. denotes at most one type.

Defn. 3.4.3 A **type identification** is a triple <N,R,T>, where N is a textual name standing for a type descriptor within the reach R, and T is an assignation interval of the space represented by <N,R>.

Observe that the assignation interval may include compile time intervals. For instance, when declaring a type at compile time, this declaration initiates an assignation interval. Notice that some languages permit compile time variables, e.g. PL/1's preprocessor. User defined compile time values occur also in so called extensible languages. For example, in ALGOL68 we are able to define new types [i.e. modes] and operators. These definitions are taken into account at compile time.

The inclusion of the reach in the type identification may be regarded as superfluous. In fact it is just a stressing that textual names may mean different things at different textual places. That is, the name map requires the reach as one of its parameters, c.f. section 2.2. In a parallel processing environment, we expect that assigning a value to a common space always occurs within a mutually exclusive section. This automatically prevents a given type identification standing for different types, since the assignation intervals are necessarily disjoint.

Following will be our definition of type equality. In the sequel, we will examine its impact on programming.

Defn. 3.4.4 Two types τ_1 and τ_2 are **equal**, iff their type identifications are equal.

Observe that a type identification stands for the descriptor, and that there could be at most one descriptor for any one type identifier. It follows immediately, that

for each type, there is one and only one type identifier, i.e. descriptor. That is, the problem of determining equality of types does no longer exist. It is also immediate that this definition of equality of types satisfies the theoretical type equality. That is, if two types are not theoretically equal they are also not equal using our concept.

The first difficulty to overcome is that of type-wise correct information transmission. That is, how can we assure that global, actual/formal and sender/receiver parameter associations do not induce parallel multityping. If there is one and only one descriptor, this descriptor must be known to each program section, e.g. module, making use of it. Of course a type descriptor could be made global within a textual portion of the program. Within this textual portion no major problems with respect to type identification will occur. This rule is not satisfactory, however, since we are either causing modules to have access to more information than they need, or we are forced to produce quite complicated scope rules. In the former case, type descriptor dissemination increases module coupling as defined by Myers[Mye1]. In the latter case, syntactic difficulties are imposed on the language.

Another form of disseminating type descriptors, is to pass along with the information itself, also the type descriptor applying to this information. Observe that this is another instance of incomplete type descriptors. That is, a formal parameter list may leave open the type of some or all of its parameter elements. Again, a textual section using this kind of incomplete descriptors could be considered a macro. This macro is then expanded for each of the actual types transmitted. It could also be considered a

function where type checking is performed whenever an undefined type operand is used. The major problem here is that "user type" is not as transparent as it was in section 3.3. This follows from the fact that here we are effectively manipulating the data of type "user type", whereas in section 3.3, we considered such data only as a static entity.

The manipulation of information is performed by means of operators. We are faced now with the problem of defining equality of user defined operators. This time we must be able to verify the equality of the operator transmitted and the operator expected. Again, we do not go far, since this decision cannot be made by means of an algorithm in the general case. We must then transmit the operators.

Within the type descriptor being sent and the receiving program section, each of the operator textual names is well understood. We could expect then, that the receiving program section uses the same textual name as that used to define the operation within the type descriptor sent. It is easy now to associate the operators bearing the same name in the type descriptor, as well as in the receiving program section. However, we are faced now with a naming problem. That is, the receiver could have been written using a different name from that in the type descriptor sent. This could occur, for example, due to the existence of a program library. We can solve this problem by the use of <u>parameter names</u>. Such names serve only for the purpose of associating a formal parameter, i.e. a receiver parameter, with an actual parameter, i.e. a sender parameter. This name has to be equal to the receiver parameter. Furthermore, the scope of such a name is restricted to the one parameter association being performed. This

brief explanation of parameter names should be sufficient for our purposes here. In chapter 5 we will study parameter transmission in greater detail.

Observe that we are using the term "parameter" in a more general form than usual. That is, the effective "value" being sent (actual parameter) is the type descriptor. However, the parameters we are refering to here, are internal to this type descriptor and are sent as a consequence of sending the type descriptor. Similarly the receiving (formal) parameter is of type type. However, the expected operations are received as a consequence of receiving the type descriptor parameter. We will denote this kind of parameters as hidden parameters. In figure 3.3.4 we have shown such a hidden parameter. The operator "equal" defined within "user_type" is transmitted indirectly when transmitting "user_type".

In the same form as we are able to solve the naming conflicts with respect to operators, we can solve other naming conflicts which may exist. For example, subfield naming conflicts. In figures 3.4.1, 3.4.2 and 3.4.3 we show, among other things, how to use parameter names to overcome naming conflicts. The construct contains(...) provides a means to make explicit the hidden parameters sent by the type descriptor. Observe that the parameter lists of type_1 and type_2 are mixed together into one single parameters list. This is possible due to the use of parameter names within this single list.

Now, the expected operator is supposed to perform a given task. We must then be able to assure that the operator sent indeed performs this task. Again we are faced with an undecidable problem. This time we must rely on the

ability of the programmer to provide an appropriate operator. This implies also, that the receiving program section must specify all operators it expects to receive in sufficient detail.

Summing up, we have then, that in order to disseminate type descriptors:

a- they could be made available through the use of an appropriate choice of scope;

b- they could be sent together with the information they describe.

In case (b) all operators expected on the receivers side must be made available by the sender. If this fails to occur, two possible actions could be taken:

i- cause a nonexistent_operator error to occur at the instant an operator, which has not been sent, is attempted to be used;

ii- cause an incomplete_descriptor error to occur, at the instant the type descriptor is received.

Another interesting aspect of sending type descriptors, is that we may restrict ourselves to send those and only those operators which will effectively be required. This has several advantages, in particular we are able to provide a greater degree of protection.

Notice that type descriptor transmission does satisfy the requirement of one and only one type definition. Furthermore, it allows virtually any program section to process information of any of the currently existing types. We have then:

Lemma 3.4.5 With the use of either scope rules or type descriptor transmission, the type equality defined in 3.4.4 corresponds exactly to the theoretical type equality.

Type adaptive operators exist both in PL/1 [GENERIC] and ALGOL68 [op definition]. Furthermore, in ALGOL60 it is not necessary to define the type of the formal parameter. In this case, the operators within the receiving procedure must adapt to the type of the associated actual parameter.

Our form of type identification and equality, is useful also with respect to independent compilation, c.f. Palme[Pal1]. If the language processor, i.e. during some compile step or at run time, is able to read a type descriptor from an external file, it enables the user to create a library of type descriptors. Observe that PL/1's %INCLUDE and SIMULA67's external class provide such a facility. The reach of such a type descriptor is now the set of all programs which are compiled and/or executed requiring this descriptor. The assignation interval of such a descriptor starts at the instant this file was last created or updated. In the case of type descriptors obtained from external files, type inconsistencies due to incorrect assignation intervals, may be corrected by means of the recompilation of the program sections using the outdated type descriptor. Observe that such a facility is provided in the PDP10 time-sharing monitor[DEC1,DEC2], with respect to source and object programs.

When receiving a type descriptor from an external file, we are usually not interested in receiving the whole text of this type descriptor. Rather, we are interested in receiving only the textual names and macros which are

defined by such a type descriptor, i.e. the interface information. The effective code of the type descriptor is necessary only when assembling together several independently compiled modules. This leads us to the conclusion that a "compilation" should produce at least following results:

i- interface information, i.e. all information required to interrelate with another independently compiled program module;

ii- partly or completely compiled code.

Some additional information could also be produced. For example, a storage layout description could be provided, such that a symbolic dump could be performed whenever required or after aborting execution.

Let us examine now what the contents of the interface information is. Interface information is refered to by some textual name. If this textual name does not explicitly define the triple <N,R,T>, i.e. the program module identification, this identification, or its encoding, must be present within the interface information. This follows from the fact that <N,R,T> is necessary to determine the equality of the program modules it identifies. Now if the textual name does explicitly define <N,R,T>, equality could be assured directly by inspection of the textual name. Usually though, interface information will be kept on some file, where this file name does not convey <N,R,T>. Furthermore, the cost of inclusion of <N,R,T> into the interface information is minimal. We will expect then, that the identification be present.

A valid encoding of the identification <N,R,T> will be called a creation stamp, where such a creation stamp is a value of some type, e.g. integer, character string or even <N,R,T>. Furthermore, at any given instant, no two equal values are present within the whole system. Creation stamps could be generated, for example, by reading a high resolution (day time) clock[Fenl], or by reading the value of a system defined counter which is increased by one for every access and is never reset. We will assume then, that all information related to some program module, e.g. interface information, code, layout description, input text, extensions etc., bears one and the same creation stamp. By simple comparison it can be established then if two different program modules refer to the same entity.

Besides the creation stamp, interface information will also contain descriptional information, and operational information. Descriptional information describes the characteristics of the information received or transmitted by the program module, e.g. parameter lists, returned value type. Operational information defines the operations which could be performed by this program module, e.g. access function, internal types, entry points. Notice that if the operation is a macro, the text of this macro must also be provided as interface information.

We have mentioned before, that, in some cases, not all operations are to be made available to the exterior. This can easily be accomplished by not including this operation into the interface information. Similarly we need not, or do not want to, make descriptional information available. For instance, we may omit that a given function returns a value without establishing an error condition. In some cases also formal parameters may be omitted if it is known

that the program module will operate correctly in their absence. This justifies then the classification of descriptional and operational information into essential and non-essential information, where essential information is loosely defined as being the subset of the descriptional and operational information which is necessary in order to allow the program module to correctly perform the required tasks. According to Myers[Mye1], non-essetial information should be absent in order to reduce module coupling.

Observe that operations made available within the interface information may require themselves interface information. Thus we have in fact a hierarchy of interface information. Due to the existence of creation stamps, we may copy the interface information of such operations. This follows from the fact that we may verify equality at some later point by comparing the creation stamps. This shows also that the creation stamp must be kept at least until the corresponding program module M is assembled into some completely containing module, where this completely containing module is such that its interface information does not contain any of M's interface information.

So far we have seen that a complete type descriptor may be transmitted. However, in many cases we are not interested in the whole data space, but rather in a subspace thereof. For example, when processing a list, we may be interested in a certain portion of a node, but not in the whole list. Let us examine following solutions to the problem of subspacing:

a- data descriptors are unique, and each program section making use of them must be in the reach of the definition;

b- subspaces may be accessed only through operators defined on the whole type;

Case (a) has already been analyzed. We have also found that the problems relative to this solution lie on the side of proper scope definitions. There are no naming problems since it is the data descriptor which governs the naming, i.e. the data descriptor does not have to adapt to an already given set of names.

Case (b) is not as restrictive as it looks. The operators defined for a given type, could be the access operators described in section 3.3. Thus access to subtypes is possible, even if types are transmitted. However, access functions deliver values of a given type. This implies that the operator sent must produce a result of a type which is known to the receiver. But since the type, and consequently its subtypes, are transmitted, we must also transmit the descriptors of all subtypes which might be required by the receiver.

The last consideration to make, is that some program section may act as an information distributing section. That is, this kind of section receives information and passes it on, without looking at it. It is clear that we do not wish to force this information distributor to know the details of type descriptors being transmitted, in particular with respect to subtypes. This could be solved by allowing type descriptors to be grouped. In this case, there will be just one type descriptor sent, however, within this descriptor, several other descriptors exist. An example of such a grouping is shown earlier in this section, where type A is defined and, in doing so, also type B is defined. Observe that this grouping is natural if

```
ref type_1 function concat(type type_1 contains(integer size;
            ref type_1 function obtain(integer length);
            type type_2; type_2 elem[*]); ref type_1 a,b);
   begin concat;
      integer length = a->size + b->size;
      concat:=obtain(length);
      for i:=1 until a->size do
         concat->elem[i]:=a->elem[i]; od;
      for i:=a->size+1 until length do
         concat->elem[i]:=b->elem[i-a->size];
   end concat;
```

Following hidden parameters have been made explicit:

```
ref type_1: obtain(length:: );
integer /*fetch function*/ size;
type_2[*] /*access function*/ elem;
```

Figure 3.4.1 Example of a function accepting type parameters.

the interface information allows for hierarchies of information.

In figure 3.4.1 we show the function "concat". This function receives parameters as well as the type descriptors of these parameters. We also show the set of operators which must be minimally provided in order to use that function. In figure 3.4.2 we show a type definition which will be transmitted to "concat" of figure 3.4.1. How that transmission is accomplished is shown in figure 3.4.3.

In figure 3.4.1 the contains(...) construct turns explicit the hidden parameters transmitted by use of "string". In figure 3.4.3 the type "string" of figure 3.4.2 is transmitted to "concat" of figure 3.4.1. The names

```
type string (integer string_size) =
   begin string;
      outside_scope scope;
         integer fetch function string_length=string_size;
         bit(2) string_elem[string_size];
         ref string function get(integer length) =
            get:=new string(length);
         ref string function constructor(bit(2) vector[*]) =
            begin constructor;
               integer length=upper_bound(vector);
               constructor:=get(length);
               for i:=1 until length do
                  constructor->string_elem[i]:=vector[i];
               od;
            end constructor;
      end scope;
   end string;
```

Figure 3.4.2 Definition of the type "string of bit(2)".

```
ref string a,b,c;
a:=constructor(array('01'B,'10'B));
b:=constructor(array('11'B));
c:=concat(type_1::string(elem::string_elem,size::string_length,
      obtain::get,type_2::bit(2)), a::a, b::b);
```

Figure 3.4.3 Example of the use of "concat" with type "string".

defined by "string" and the names expected by "concat" are different. This difficulty is overcome by means of the parameter name association. The syntax used is:

<formal parameter name> :: <actual parameter>

where <formal parameter name> is the textual name of the formal parameter which will be associated with the value of <actual parameter>. The construct array(...) in figure 3.4.3 builds a vector of as many elements as there are actual parameters in the list. The elements of the array are initialized to the actual values of the parameters.

Earlier in this section, we mentioned that users should be enabled to define different types, based on the same definition. Recall that this is necessary to solve problems relative to semantic type inequality. It should be clear now, that this could be achieved simply by defining several textual names for the same base type. Due to the type equality defined, these types are necessarily different. In the example shown, we would have then:

```
type student=char(40);
type professor=char(40);
```

Both definitions rely on the same base type. Since no operations are provided explicitly, the standard base type operations apply. However, due to the different textual names, both types are different. Furthermore, there is no conversion defined between them, thus these two types are necessarily disjoint.

Theorem 3.4.6 Let $\langle \alpha, \tau_S \rangle$ be a data space being accessed by an access path $\langle b, \tau_P \rangle$. The access is type-wise correct, iff the type identification of τ_S and τ_P is equal.

The proof of this theorem is an immediate consequence of the preceding discussion. Observe also that it makes use of our type equality definition. Thus, the only if portion may fail, if another equality definition is used.

Observe that across module boundaries different textual names could refer to a same type. If we would take our definition strictly we would have two different types in fact, although we are defining only one. This shows then, that we must allow renamings of textual names to occur across module boundaries, without that new type descriptors are created. Notice that, by means of parameter names, these renamings define an equivalence class of textual names, where this equivalence class is defined with respect to the possible renamings across module boundaries.

Let us examine now some of the main differences and similarities between our type descriptor system and SIMULA67's classes.

a- Both systems rely on uniqueness of definitions.

b- In our system data belonging to a type descriptor is made available by means of a specific declaration. In SIMULA67 local data of classes is available to all blocks to which this class is prefixed. Thus our system offers more control over data, easing thus the task of certifying a descriptor†.

c- The class hierarchies which can be built in SIMULA67 are strictly tree like hierarchies, where each class appears exactly once as a node in this tree. In our data descriptor system, there is no restriction in which way to combine different type definitions. Our system is thus more general. Whether or not this increased generality is definitely required, will not be discussed here.

---

† In a recently proposed modification of SIMULA67[Pal2] a new attribute, i.e. hidden, is defined. Names possessing this attribute are prevented from being accessed from the exterior of the class definition.

d- Our type definitions may be extended at will, provided that the scope and protection restrictions are obeyed. Class extensions exist also in SIMULA67 in an embrionic way though. That is, we must provide all textual names which might be used to extend a class definition, by declaring them virtual. Since SIMULA67 does not protect local information, this represents no major restriction.

e- We may transmit our type descriptors and their extensions, in order to disseminate type definitions. In SIMULA67 classes are made known exclusively by means of ALGOL-like scope rules. We gain again in a more controlled environment. Furthermore, we are creating an environment where data and its operational aspects may be disseminated among several machines. Whether there are major restrictions to this "network" problem, will not be discussed here.

3.5 Type Checking.

In the previous section we have seen that type-wise correct access is guaranteed if both, access path and accessed data space, imply the same type identification, i.e. type. We will study now how we could enforce this rule statically, and how we could enforce statically that data spaces are sequentially multityped. First we have to obtain several results which are valid both for the dynamic as well as for the static case.

Theorem 3.5.1 Let $\langle \alpha, \tau_s \rangle$ be a data space. Let t be the instant when α is typed τ_t, where $\tau_s \neq \tau_t$. Let T_t be the τ_t typing interval starting at the instant t. α is

sequentially multityped, iff all accesses to α within T_t imply τ_t, and:

a- the first access to α within T_t is a write access; or

b- there is a conversion C_{st} which is defined for the value of type τ_s in α at instant t. Furthermore, this conversion is performed during the typing of α with τ_t, and no other access than those perfectly understanding this typing and conversion are made to α.

Proof. By theorem 3.4.6 it is immediate that any access to α within T_t must imply τ_t.

If condition (a) holds, then the contents of α are necessarily of type τ_t after the access. Furthermore, they are irrelevant during the interval starting at the instant t and ending immediately before the write access to α. That is, the contents of α are undefined during this interval.

If condition (b) holds, it is immediate that, after completion of the typing operation, any access implying τ_t causes a type-wise correct access to occur. Furthermore, accesses during the typing operation are type-wise correct, since the typing is well understood.

Suppose now that both (a) and (b) do not hold. Thus the value in α is not implicitly undefined. Furthermore, τ_s and τ_t are different and the typing of α with τ_t is not completed when a τ_t implying access occurs. But then α is accessed implying τ_t when the contents of α are

not yet of type τ_t. Therefore α is not sequentially multityped, contradiction■

A consequence of this theorem, is that error interrupts ocurring during typing as defined in condition (b), must well understand the typing operation, or then must be restricted in their access capabilities. Of course, we could imagine an identity conversion and produce a dump when such an error condition occurs. However, it is our feeling that dumps of this sort should be the last resort. Thus we would like to see why and how an error occurred in a form which resembles the data being processed, i.e. a symbolic dump. This suggests then, that the error handlers for conversion and/or typing operations should belong to the data descriptor involved in the conversion and/or typing.

Observe that if the conversion C_{st} is an identity conversion, there is no explicit conversion performed when typing α. That is, there is no code which performs this conversion within the program. This fact may be used, in principle, whenever the type set T_s is a subset of T_t. However, if T_t is a proper subset of T_s, a membership test must be performed by C_{st} in the general case. That is, C_{st} is at most a partial identity conversion. Thus, the conversion appears within the program. Whether such a membership test can be avoided is again an undecidable question, since answering this question corresponds to determining the equality of two functions, i.e. the type τ_s value generating funtion and the conversion predicate.

Suppose now that there is a subspace β of α, which is typed τ_b by both τ_s and τ_t. Thus there is an identity conversion with respect to this subspace. Consequently

condition (b) of theorem 3.5.1 holds trivially with respect to β. Thus, access to such a data space is always possible, even if it is not valid for the whole space α typed by either τ_s or τ_t. Generalizing and formalizing, we have:

Lemma 3.5.2 Let α be a data space. A subspace β of α being sequentially multityped does not imply α being sequentially multityped. Nor does a failure of the conditions in theorem 3.5.1 imply that a subspace β of α necessarily fails these conditions too.

When typing a data space α, a subspace β of α may be in either of the following three states: "not yet typed", "being typed" and "already typed". It should be clear that the typing of α must be understood in order to access β. However, accesses to β possess different properties than those to α. β could then be accessed, even if α has not been completely typed, without incurring in type-wise incorrect accesses. To solve these problems mechanically may require a reasonable amount of run time checking. Furthermore, these problems are common to the problems of multiple access paths.

Let b be an access path accessing $\langle\alpha, \tau_s\rangle$. Suppose now that b implies a type τ_p, such that there are identity conversions I_{sp} and I_{ps} defined between τ_s and τ_p. As we have already mentioned, an identity conversion could be an implicit conversion. If this is the case, b accesses α as if α were of type τ_p. Apparently this violates lemma 3.4.6 and theorem 3.5.1. In fact it does not, due to the implied conversion. That is, any access to α by b may be regarded as implying τ_s. We have also mentioned that the user has to provide the conversion functions, and also that he has to decide when to use such a conversion. Of course some of the

work could be reduced, by an appropriate choice of coercion rules as in ALGOL68.

Both identity conversions I_{sp} and I_{ps} are not always required. This stems from the fact, that, depending on the access form of the access path b, i.e. read only or write only, only one of the identity conversions must exist. However, the conversion must always be impliable, i.e. the domain type set must be a subset of the range type set. Formalizing we have then:

Lemma 3.5.3 Let $\langle \alpha, \tau_s \rangle$ be a data space. In a sequentially multityped environment, a τ_p implying access path b may be bound to α, iff:

a- there is an identity conversion I_{sp} defined for all values of type τ_s, if all accesses via b are read accesses; or

b- there is an identity conversion I_{ps} defined for all values of type τ_p, if all accesses via b are write accesses; or

c- both I_{sp} and I_{ps} exist, if accesses via b may be either read or write accesses; or

d- any access via b satisfies the conditions of theorem 3.5.1.

Observe that condition (d) means that α may only be accessed via b, if α is currently of type τ_p. It does not mean that binding b to α is forbidden if α is not of type τ_p. Observe though, that the determination of whether b will access α outside of a τ_p typing interval, is undecidable in the general case.

We want to stress again that this lemma does not contradict theorem 3.5.1, since any b access is implicitly preceded by a conversion. Furthermore, we want to stress

the fact that it is undecidable whether a given identity conversion I_{ab} can be implied if the type set T_a is a proper subset of T_b. From this we have that such an identity conversion should not be implied, if "correct" results are to be proved mechanically.

We did not mention union types so far. In the case of "universal union" types, theorem 3.4.6 always applies, since the type of the data space is always the same. Consider now the case of selective unions, e.g. ALGOL68 union types. It should be clear that any single type could represent a union type, where the effective type set is of cardinality one.

Let τ_a and τ_b be two union types. Let $\tau_{a1},\tau_{a2},\ldots,\tau_{an}$ be the effective types of τ_a. Let $\tau_{b1},\tau_{b2},\ldots,\tau_{bm}$ be the effective types of τ_b. Let $\tau_{c1},\tau_{c2},\ldots,\tau_{ck}$, $k\geq 0$, be the set of types in the intersection of the effective type sets of τ_a and τ_b. Thus, if $\langle\alpha,\tau_a\rangle$ is effectively typed τ_{ci} for some $1\leq i\leq k$, it is immediate that there could be an identity conversion between τ_a and τ_b, and vice-versa. These identity conversions do not necessarily exist, since the types τ_a and τ_b could have been implemented in a different way. For example, the type descriptor defining the current effective type could be placed differently in store or have a different encoding.

We say that the implementations of two types are equal, if their type spaces are equal, and the implementations of all subtypes are equal. Notice that this definition is basically the same as that of theoretical type equality, only that here we are not interested in the type sets and the operator sets. It follows from this definition, that the relative positioning of each subtype,

within a contiguous space must be equal. It follows also that the encoding of the effective type descriptors of a given union types must be equal. Now, if we fix a given effective type which is common to two union types, as long as the encoding of this effective type is equal and the remaining fields of the union types are equally implemented, the two data spaces are equally implemented. It follows then, that some canonical form of type encoding should exist, since otherwise the implementational equality could not be verified. For example, types could be encoded by assigning ordinal integers to each descriptor as they are defined. For practical purposes, such an encoding is unlimited. Furthermore, this encoding is viable, since each type is defined once and only once.

Formalizing we have then:

Lemma 3.5.4 Let τ_a and τ_b be union types having $\tau_{c1}, \tau_{c2}, \ldots, \tau_{ck}$ as the intersection of their effective type sets. The identity conversions required by lemma 3.5.3 exist, iff, when fixing τ_{ci}, the implementations of τ_a and τ_b are equal.

Observe that τ_a and τ_b are not necessarily equal, since each may have a different effective type set. However, they trivially allow identity conversions for all values of type τ_{ci}. Of course, it might be desirable to prevent such conversions, in order to assure semantic type inequality. This can still be accomplished, by defining the corresponding effective types to be different.

In ALGOL68 the "unite" coercion rule observes lemma 3.5.4. However, the set of effective types of the receiving data space, or access path, must contain the set of effective types of the sending data space. The reason for

this, is that type checking is performed statically in ALGOL68.

We will study now the conditions which must be satisfied in order to perform static type checking. The importance of static type checking arises from the fact that the over all cost to produce, compile and execute a program may be greatly reduced in some cases. It does not necessarily have to be so, since some hardware, e.g. the Burroughs B6700, could perform part of the type checking. Such machines offer a limited scope of type checking though, since user defined types are usually not checked. Thus, even in such machines, the problem of software type checking arises, when types are considered in our generalized form.

The execution cost is reduced, since a given access path is known to imply the correct type at run time. The compile cost is increased, but usually by a smaller amount than the reduction of run time cost. Thus usually there will be a positive cost differential. This cost reduction is usually due to the necessity of repeating redundant type checkings at run time, whereas the type checking would be performed exactly once if done at compile time. Finally several persons argue that declaring all variables and their types, increases the probability of a newly developed program to be correct, c.f. Kerningham and Plauger[Ker1], Hansen[Han3], Hoare[Hoa7]. We will not study these cost aspects though.

Suppose now that we could perform static type checking on dynamically declared textual names. Suppose furthermore, that, when starting execution, no partitioning of the set of dynamic textual names is known. Due to our assumption of

static type checking, we know that a dynamic textual name's locator function implies type τ_p on the data space being accessed. Suppose now that another dynamic textual names is made known. The type implied by this new textual name must also be τ_p, since otherwise we would have to know a typing partition at compile time. Thus total lack of information about dynamic textual names at compile time, implies that all dynamic textual names stand for values of the same type, if these dynamic textual names are to be statically type checked.

Suppose now that, at compile time, we are able to partition the set of all possible dynamic textual names into sets. For each partition, all dynamic textual names in such a set imply the same type. Such a partitioning could be based on some syntactic rule, as for instance, the first letter rule in FORTRAN, i.e. the IJKLMN rule. However, the operation of determining dynamic textual name membership is now also a type checking operation, since the former implies the latter. Formalizing we have then:

Lemma 3.5.5 Static typing of dynamic textual names is possible, iff the type implied by the locator function of such a name is one and the same for all such textual names■

Observe that in some cases we could have run time compilation of some program sections, e.g. the CODE and EVAL functions of SNOBOL4. In such a case, it should be clear that "dynamic" type checking is performed for newly introduced textual names. However, once the text has been compiled the type checking could be performed statically. This does not contradict lemma 3.5.5, since, after compilation, the code behaves as if it had been compiled at

the original compile time. Furthermore, during this additional compile time, type checking is effectively performed.

A consequence of lemma 3.5.5 is that the type of all textual names has to be declared at compile time. This declaration could be explicit, e.g. ALGOL68, or implicit, as in some cases in FORTRAN and PL/1. There could also be a construct which associates a type with the set of all possible dynamic textual names, or some subset thereof. Thus in fact we are declaring the type of each of the dynamic textual names at compile time. Of course, we do not need to know exactly which are the textual names which are going to be used, nor do we impose any restriction on the effectively used set of dynamic textual names. Formalizing we have:

Corollary 3.5.6 In order to achieve static type checking, the type of all textual names, static or dynamic, must be declared at compile time.

Let $\langle \alpha, \tau_b \rangle$ be a data space, and T_b a τ_b typing interval of α. By theorem 3.5.1 any access to α during T_b must imply τ_b. During a specific T_b a given set of instructions of the program is executed. This set is not necessarily contiguous. Furthermore, if some of the input values were different when execution of T_b starts, a different set of instructions could have been traced. Not only this, in fact T_b could also have been modified, i.e. shortened or lengthened in time.

Let T be some time interval, we will call a trace, the set of all instructions, i.e. code words, which are executed during T. We must make the concept of trace code dependent, since it is not guaranteed that a higher level

statement will be completely executed during some elaboration.

Let $\langle\alpha,\tau_a\rangle$ be a data space. We will call a τ_a <u>code typing section</u> the union of all traces TR_a, such that there is an elaboration E and a τ_a typing interval T_a, during which E traces TR_a. That is a τ_a code typing section of α, is any code section during which α could be accessed by means of an access path implying τ_a. Observe that code typing sections do not necessarily consist of a single contiguous portion of code. Furthermore, several different code typing sections may have a non null intersection. Finally, code typing sections could vary dynamically, i.e. the code comprising a code typing section is not necessarily known at compile time.

<u>Theorem</u> 3.5.7 Let $\tau_1,\tau_2,\ldots,\tau_n$ be the types which could be associated with the sequentially multityped data space α. α can be statically type checked iff:

a- all code typing sections TT_i can be determined statically; and

b- all code typing sections are mutually disjoint, i.e. $TT_i \cap TT_j = \emptyset$ whenever $i \neq j$; or the identity conversions required by lemma 3.5.3 exist.

Proof. If all code typing sections are mutually disjoint, it is immediate, that only one type is associated with α during the execution of such a code typing section. Recall that the identity conversions are implied, whenever the conditions (a) through (c) of lemma 3.5.3 are satisfied. Furthermore, the code typing section currently being executed determines the type of α. Therefore, code typing

sections must be determined statically, since their determination implies the typing of α.

Suppose now that two different code typing sections TT_i and TT_j meet in TT_{ij}. Suppose, furthermore, that the identity conversions between τ_i and τ_j required by lemma 3.5.3 do not exist. Then, whenever TT_{ij} is executed, any access to α implies either τ_i or τ_j. But since α is to be sequentially multityped, τ_i and τ_j must be equal by theorem 3.4.6, contradiction■

Observe that FORTRAN as well as PL/1 violate this theorem, c.f. EQUIVALENCE, COMMON and EXTERNAL. This accounts then for the fact, that type incorrect accesses are possible in both these languages.

An immediate consequence of theorem 3.5.7, is that whenever control passes from a code typing section TT_i to another TT_j, the typing conditions of theorem 3.5.1 must be satisfied, unless the required identity conversions exist. We have thus <u>code typing section origins</u>. These origins are portions of code which effectively generate a valid value of the type implied by this code typing section. Hence, such origins satisfy the conditions of theorem 3.5.1. Formalizing we have:

<u>Corollary</u> 3.5.8 Let τ_i and τ_j be two types, such that the identity conversions described by lemma 3.5.3 do not exist. Transfer of control from the code typing section TT_i to TT_j of a data space α, is possible, only if this transfer of control passes through an origin of the code typing section TT_j■

Observe that all considerations with regard to identity conversions are important, only if there are coercion rules which may cause such a conversion to be implied. Furthermore, it should be clear that such a coercion could only include natural conversions. For example, an identity conversion between integers and reals is usually not natural.

With regard to static type checking, we conclude then that a textual name posses also a typing reach. That is, the reach of the textual name is partitioned into one or more sub-reaches, where each of these sub-reaches implies exactly one type. Such a sub-reach is then transformed into a code typing section. Usually the typing reach corresponds to the reach itself. However, this is not necessarily so, recall, for instance, the EQUIVALENCE example in section 3.2.

We have seen that if we desire to statically type check, the code has to be statically determined. Now, when processing a dynamically type checked space, this is usually done by directing control to the appropriate procedure capable of processing the current effective type. Within this procedure the type of the data space is statically type checked. For example, in ALGOL68 we have the conformity relation, which establishes the current type of the union typed data space, and may be used to direct control flow. Similarly, Hoare[Hoa5] suggests a type selector case statement. This statement directs control to substatements which assume the data space as being statically type checked. Finally, in languages like SNOBOL4 and GEDANKEN[Gri6,Rey2], each effective processing of a given data space, is preceded by a routing procedure, wich

directs control to the appropriate statically type checked data handling procedure.

4. Access Functions.

In chapter 3 we have introduced type descriptors. We have mentioned also that internal spaces of some module M, e.g. type descriptor, may be accessed by means of some access function defined by M or global to M. Such access functions, or locator functions, could occur in a variety of forms and degrees of complexity. For example, some access functions could be made responsible for traversing a tree in some predefined order. Others could be made responsible simply for accessing dynamic storage spaces. Finally, others could perform checking operations in order to determine whether a given procedure may effectively access the data space in question. We may say then, that access functions serve at least for following two purposes:

i- to characterize the storage behaviour of a given type, e.g. tree, stack, buffer etc.;

ii- to protect the data spaces internal to the modules defining the access function.

Access functions, or locator functions, could take the degenerate form of just loading an address into some register. Other access functions could compute such addresses by means of some user provided parameter. Finally, others could receive or establish control information from which these addresses could be abstracted.

When performing list processing, usually some pointer, or reference, is used to refer to a particular element of the list. It has been recognized though, that the generality of pointers may cause the certification of programs to become quite difficult[Hoa7]. This suggests then, that an

alternative for pointers be found, where this alternative should:

a- not restrict the power of expression of the programming language, nor cause the textual dissociation of related entities;

b- restrict as much as possible the set of data spaces able to be accessed;

c- hide as much as possible the implemetation of modules which make internal spaces available to the exterior;

d- allow verification of access rights.

We will show in this chapter that these objectives can be met. As a result we show that several language constructs become necessary. However, we will not study whether some of these constructs are necessarily inefficient.

In this chapter we will also examine how successive elements of a given ordered set could be generated. It should be noted that these sets or, better, sequences could be defined in a computational manner, e.g. the sequence of nodes visited when traversing a tree in some order. We will examine then the constructs which are necessary in order to implement generator functions. The concept of generator functions has been defined for a long time already, e.g. IPL-V[New1,Gel1]; it has received little consideration in present programming languages though. It will be shown that by means of generator functions the implementational characteristics of a module could be hidden, thus contributing significantly to the interchangeability and flexibility of modules.

Another problem which arises when information is interchanged between modules is that of privacy and access path control, e.g. reference count. It will be shown that, in order to overcome these problems, we must be able to redefine or extend language processor defined operators, in particular the store operator. Of course, these access controls are weak in the sense that the language conventions must be obeyed, since, otherwise, the controls become ineffective.

In this chapter we will frequently use the term "access function" to denote "locator function" or even "access path". This should not be thought as an abuse of nomenclature, since access functions are defined in terms of locator functions which, in turn are defined in terms of access paths.

This chapter is subdivided into 5 sections. In section 4.1 we study locator functions, i.e. the set of language processor defined access functions. In this section we determine how language primitive types and operations could be implemented. We show also in this section some basic results which will be used in later sections. In section 4.2 we study the implementation of dynamic data spaces. The main objective of this section is to determine the constructs required for the implementation of storage managers. A secondary objective is to determine the conditions under which these constructs will maintain a sequentially multityped environment. In section 4.3 we show how pointers could be outlawed without violating the conditions stated above. For this purpose we introduce the concept of data space identifications. In section 4.4 we study how to control the dissemination of information externally to the module generating the information. We study also the form

in which language processor defined operations could be replaced by user defined functions. Finally, in section 4.5 we study a class of functions which allow us to gain access to data spaces without requiring data space identifications to be provided as external parameters. By means of these functions we are able to hide away the implementation of modules.

4.1 Locator Function Parameters.

In section 2.1 we have defined locator functions as being a composite function corresponding to a name map and the access path delivered by an elaboration of this name map. We have mentioned also that, in order to associate a unique data space with a textual name, the evaluation of the locator function may require parameters. Following are the kinds of parameters used by locator functions:

i- actual/formal- parameters of this kind are external to the locator function, and are provided by the user's program. Examples are: subscripts of arrays; qualifiers of structure elements; pointers to dynamic storage spaces.

ii- intrinsic- parameters of this kind are part of the data space being accessed. Examples are: bounds of dynamic arrays; variables used by the REFER option in based structures in PL/1; the effective type descriptor field of union typed spaces in ALGOL68.

iii- local- parameters of this kind are part of the locator function's space. Examples are: bounds of fixed size arrays.

iv- global- parameters of this kind are part of the environment within which the locator function exists. For example, ALGOL60 display vectors.

In section 3.3 we have mentioned that access functions must be defined for each of the existing types. Since we are unable to predict all the types which will be used, we conclude that programmers must be able to define some of the required access functions. An immediate consequence is that there may be incorrect access functions. We will assume, though, that user written access functions have been certified by some means.

The term actual/formal parameter is actually a misnomer. That is, the form with which such parameters are made known, does not always resemble the conventional form of a parameter list. For example, in the SIMULA67 construct:

OBJREF•OBJVAR †

OBJREF is an actual parameter required by the "locator function" of OBJVAR.

Actual/formal parameters may be provided in an implicit way. For example, when accessing an element of a PL/1 structure it is not necessary to provide always the complete qualification list of this element.

Usually actual/formal parameters do not cause side effects. However, consumption of the actual parameter is an example of such a side effect. For example, when access is based on some form of matching, frequently the matched heading portion of the actual parameter value is consumed, i.e. deleted. If the matching succeeds, the actual parameter value is set to empty. Otherwise, the actual parameter

---

† We use the character "•" to denote the remote identification operator "." of SIMULA67.

value reflects the not matched portion of the original value.

When passing information to a locator function, this information could be incorrect and, thus, cause the locator function to fail or to deliver an unexpected name. This may occur even if the locator function itself is correct. This suggests then, that parameters should be tested for validity when being received by the locator function.

The next few paragraphs will apply to all kinds of parameters. The results are more crucial with respect to actual/formal parameters though. We will use the term "actual parameter" to denote the information sent to the locator function. The term "formal parameter" will be used to denote the information expected by the locator function.

Formal parameters are of some type, say τ_f. Intuitively we would expect that every element in the type set TS_f is a valid parameter value. This is not so, though. Consider for instance an ALGOL60 array. The locator function which accesses elements in this array, accepts formal parameters of type "integer" specifying which element is to be accessed. However, not all "integer" values are valid elements.

Defn. 4.1.1 The exact formal type of a formal parameter of a locator function F, is a type τ_e such that every element in the type set TS_e is valid, and every valid element is in TS_e.

The first difficulty which shows up, is that exact formal types may be dynamically defined. Furthermore, the exact formal types may depend on the values of several other formal parameters. For example, let A[N,N] be an

upper triangular array. It is now immediate that, for an access A[I,J], the following relations must hold: 1≤I≤N and 1≤J≤I. Figure 4.1.1 shows a possible definition of such an array. Observe that this is an example where a given exact formal type, i.e. that of index J, is dynamically defined and depends on another parameter, i.e. I.

```
type upper_triangular .(integer N) =
   begin triangular;
      real array matrix(N*(N+1)/2);
      outside_scope operations;
         real access function .(integer I,J)=
            begin access;
               if (1≤I≤N)&(1≤J≤I)
                  then matrix(I*(I-1)/2+J);
                  else fail;
               fi;
            end access;
      end operations;
      matrix:=0.;
   end triangular;
```

The monadic operator "." stands for "concatenate with identifier". This allows writing a declaration list, as well as using empty (access) function names.

Example:

```
upper_triangular A(20), B(20);    /* declaration*/
A(3,1):=B(2,2)+10.;
```

Figure 4.1.1 Type "upper_triangular" with explicit checking of locator function parameters.

Defn. 4.1.2 Parameter checking is the operation of determining whether the value received by a formal parameter is of the exact formal type.

Of course, the type of the actual parameter sent will be equal to the type of the corresponding formal parameter. However, not always will the formal parameter type be equal to the exact formal type. We will say that an actual/formal parameter association is *equal type checked* whenever the type of the formal parameter is also the exact formal type. In PASCAL the ranges of an array are defined by means of a scalar type, e.g. a contiguous subset of the integers. Each element of this subset is a valid index of the array. Thus, the parameters to the locator function which accesses an element of an array are equal type checked in PASCAL.

If the formal parameter type is not the exact formal type expected by the locator function, *explicit parameter checking* must be performed. Of course, this checking could occur at a conceptual level. In the example shown in figure 4.1.1, the parameters to the access functions known externally, i.e. I and J, as well as the parameters known to the internal locator function accessing an element of "matrix", are explicitly checked. Explicit checking does not always imply run time effort. For example, the internal access to "matrix" in figure 4.1.1 is valid for all values of I and J, as can easily be proved. The access correctness proof of this particular locator function is an example of explicit parameter checking occuring at a conceptual level, i.e. externally to the language processor. This suggests then, that programmers should be able to inhibit the automatic inclusion of explicit parameter checks. We may conclude then:

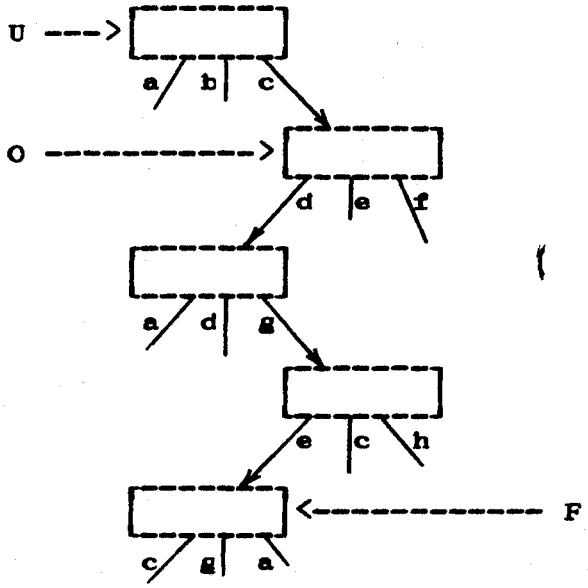
Fact 4.1.3 Dynamic explicit parameter checking may be redundant, even if the parameters are not equal type checked.

We will examine now whether there is a locator function for which equal type checking of parameters is insufficient to assure receiving values of the exact formal type.

Let U be a tree with labelled edges, and O some distinguished node in U. Let s be an ordered sequence of labels separated by the symbol ">". Access to a node F in U is now defined as follows:

a- starting at node O follow a path in U, such that the labels of the edges in this path are the same, and occur in the same order, as those in s. If such a path cannot be found, the access fails.

b- F is the node reached by the edge corresponding to the last label of s.



For following t:

s=d>g>e

access to node F below will occur.

Figure 4.1.2 Access to MULTICS files by means of absolute or relative paths, a simplified approach.

Figure 4.1.2 shows a diagram of such an access mechanism. This access mechanism is a simplified version of the MULTICS file access mechanism. We have simplified this

access mechanism by not including protection mechanisms such as passwords and ring levels[Mad1,Org1]. This same file access mechanism is implemented also by the Honeywell 6000 GCOS operating system.

Notice that for file systems in particular, redundant operations should be avoided, in order to decrease the amount of information traffic between secondary storage and main storage. Notice furthermore, that this example could be an instance of the actual parameter consumption example mentioned earlier in this section.

Let U be a subtype of the type "file_system". It can easily be shown that the access mechanism above can be defined as an operation within "file_system". It can also easily be shown that U itself could be maintained local to the type definition "file_system". Now, checking the validity of a path s, corresponds to verifying if there is a path in U, starting at O, which traces the labels of s. Thus, checking the validity of s corresponds exactly to the operation of gaining access to F. Thus, in this case, performing parameter checking separately from the actual locator function elaboration is redundant.

The type set TS_e of the exact formal type τ_e of s, is the set of all valid sequences of labels for all valid origins O. However, the information content of U is internal to "file_system", thus, outside of the type "file_system", it is impossible to define the exact formal type τ_e. Thus, the actual parameter must be of a type, say τ_a, such that the type set TS_e of the exact formal type τ_e, is a subset of the type set TS_a. Since the type of the actual and formal parameters must be equal, and conversion cannot take place outside of the type "file_system", the

formal parameter must also be of type τ_a. Formalizing we have then:

Theorem 4.1.4 There is an access mechanism for which equal type checking cannot be performed.

Observe that if the type of the formal parameter is equal to the exact formal type, and if also the conditions of theorem 3.5.7 [static type checking conditions] are met we are able to perform equal type checking at compile time. Thus, equal type checking of parameters may allow some of the checking to take place at compile time. However, it does not imply that all checking takes place at compile time, since now the type sets are more precisely defined and, consequently, checking will be performed within conversion routines. It is argued, however, that execution time cost would be decreased, since less actual checking would be performed at run time when equal type checking is enforced[Wir2,Han3,Hoa5].

Observe that in order to perform equal type checking, we must be able to prove that the operations performed on values of a given type, preserve the characteristics, e.g. type set, of this type. For example, we must be able to prove that subscript expressions generate values within the range of the corresponding array index. It follows immediately that a compiler cannot perform this proof in the general case. Not only this, the number of cases where the compiler is able to produce such a proof is relatively small. Furthermore, if the programmer is able to provide the necessary proofs himself, it seems to be a waste of machine effort to perform explicit checking. We may conclude then, that in practice there is no definite gain when forcing all parameter checking to be equal type

checking[Hab2]. Consequently, there should be language constructs which allow (privileged) programmers to suspend explicit parameter checking in order to increase the efficiency of a program. As long as correctness proofs have been produced externally, this offers no risk of incorrect computations.

```
type index .(integer bound)=integers(1...bound);
type upper_triangular .(integer N) =
   begin triangular;
      real array matrix(N*(N+1)/2);
      outside_scope access;
         real access function .(index I(N), J(I)) =
                     matrix(I*(I-1)/2+J);
      end access;
      matrix:=0.;
   end triangular;

Examples:

   integer constant N=10;
   upper_triangular C(N);
   index l(N), k(l);
   C(l,k):=1.;     /* i.e. C(I::l,J::k,N::N):=1. */
```

Figure 4.1.3 Definition of "upper_triangular" where parameter checking can be performed by equal type checking.

In figure 4.1.3 we show how the type "upper_ triangular" could have been implemented using equal type parameter checking. For this purpose a global type "index" has been defined. The type set of "index" includes all integers from 1 to some specified bound≥1. Equal type checking is possible, since the actual parameter l and its corresponding formal parameter I are of the same type, i.e.

"index(N)". Furthermore, the types of the actual parameter k and the corresponding formal parameter J are also equal, i.e. index(l). Thus, except for renaming, the actual and formal parameter types are equal. This renaming could have been made explicit by means of parameter names as shown within the comment. The bounds of the type "upper_ triangular" can be left open, since a global named constant is being used to declare the bound of "upper_triangular" as well as that of the types "index" used externally to "upper_triangular".

Observe that this is not a contradiction to our type equality definition in section 3.4. "Index" has been defined exactly once. Since the constant "N" is global all instances of "index" using this "N" can be viewed as being renamings of a globally defined instance of "index". Of course we are assuming that the language processor is able to verify these renaming conditions. As has already been mentioned, the language processor could be aided by means of parameter names. Finally, observe that any assignment to l or k will be preceded by a test to determine whether the value being assigned is indeed an element of the corresponding type set. That is, we have traded the explicit checking for a conversion predicate elaboration.

Locator function parameters can be divided into two major groups:

i- descriptional parameters, which describe properties about the data space at hand. For example, array bounds; effective types of union typed spaces; structure or record layout.

ii- positional parameters, which provide names, or

parameters to access paths to other spaces. For example, list pointers; array base addresses.

Descriptional parameters can be determined statically or dynamically. For instance, the layout parameters of a PL/1 structure not containing variable array bounds can be determined completely at compile time. Thus, the explicit values of the layout parameters could be discarded after code generation has been completed. Now, if a PL/1 structure contains dynamic array bounds, the values of the layout parameters cannot be computed at compile time. Thus, at least partially, the layout information must be kept at run time.

In the same way as descriptional parameters, positional parameters could also be determined statically or dynamically. Linked list pointers are an example of dynamic positional parameters. Addresses of FORTRAN IV COMMON areas are an example of statically defined positional parameters. Again, notice that it is not necessary to keep the explicit values of static positional parameters, since they could be included directly into the code.

Local parameters are part of each individual locator function. Thus, if the local parameter does not change during execution, several copies of the value might be generated. This allows then, that such values be incorporated into macro expansions of locator functions. On the other hand, intrinsic parameters are part of the data space being accessed. Thus, in a sense, intrinsic parameters are local parameters, where a different instance of the parameter may exist for each accessible data space.

A parameter being local, or intrinsic, should not mean that it cannot be shared by several locator functions, i.e.

in a sense being global. That is, if some of the intrinsic, or local parameters, of several locator functions are found to be equal, these parameters could be shared by these locator functions. Sharing could be "forced" by means of a construct such as LIKE in PL/1. The sharing of local, or intrinsic, parameters could occur also due to "inheritance". For example, when passing an array in ALGOL60, the receiving formal parameter must leave open the dimensionality and bounds of the array received. Similarly in ALTRAN a variable of type "algebraic" may inherit the layout description (set of indeterminates and their admissible powers) of some other variable of type "algebraic" by means of an assignment operation.

Certain language implementations may opt to treat local parameters as if they were intrinsic parameters. For example, static array bounds could be treated as local parameters, however, the language implementation may treat such static bounds as intrinsic parameters for a matter of homogeneity of generated code. From this discussion we may conclude, that the boundaries between global, local and intrinsic parameters are quite ill defined.

We will call <u>simple data space</u> a data space which space set contains exactly one element, i.e. space. Examples are: "integers"; ALGOLW records; PL/1 structures.

<u>Defn</u>. 4.1.5 Let α be a simple data space. Let β be a simple data space which is a subspace of α. An <u>access refinement</u> of α is an access path to β, which computes the name of β by means of a displacement from the address of α.

We will call α the <u>base space</u>, and β the <u>refined space</u>. Examples of access refinements are: access paths to

fields of PASCAL records; access paths to rows in ALGOL68 arrays. However, slices of arrays in ALGOL68 are not examples of access refinements, since a slice is usually not a simple data space. We have defined access refinement in terms of simple data spaces in order to emphasize that several access path elaborations have to be performed for non simple subspaces. Observe that access refinements imply computationally generated name values.

It should be noted here that the operations of type concatenation and union [see section 3.3] allow the automatic creation of access refinements[Hoa5,Gri1,Sta1,Den2, Knu1,Tur2].

Let α be a simple data space consisting of following contiguous simple subspaces $\alpha=\alpha_1\alpha_2\ldots\alpha_n$, where $n\geq 2$ and each of the subspaces α_i possesses a non null extent. Suppose now that the name of some subspace α_i could be obtained other than by elaboration, or pre-elaboration, of some access refinement. Thus the name a_i of α_i must be obtained without taking into account the names of α and all α_j such that $i\neq j$. But then the contiguity of $\alpha_{i-1}\alpha_i\alpha_{i+1}$ cannot be assured. Thus α is a simple space merely by accident. Formalizing we have:

Theorem 4.1.6 Access paths to proper subspaces of a simple data space require the elaboration of an access refinement.

Observe that such an access refinement could be elaborated a finite number of times yielding a list of names. Thus, when accessing a subspace, this list could be examined, and access to a particular subspace would be gained directly. That is, access to the subspace is gained by using an already existing name typed value. An example

of such a pre-elaboration is the classical way to implement ALGOL60. Here addresses are composed of two parts, one allows to gain access to a base space by means of an element in a display vector. The other part is a displacement relative to the origin of this base space and both are pre-elaborated at compile time.

Suppose now, that we would allow users to construct their own access refinements. It is well known that, in the general case, we cannot determine mechanically whether user written functions are correct. It follows then, that incorrect user written access refinements may exist, where these access refinements compute incorrect name values. This suggests then, that user defined access refinements should be prevented. Let us examine now, whether this would represent a restriction.

We will use garbage collection procedures as a counter example[Bae1,Bra1,Coh1,Goy1,Knu1,Mar1,Tho1,Pec2].

Defn. 4.1.7 An access set is a set of $n \geq 1$ access paths $\{a_1, a_2, \ldots, a_n\}$.

Defn. 4.1.8 A data space α is a garbage space with respect to an access set A, if α cannot be accessed by any combination of access paths in A.

We are using the term "garbage space" in a slightly different fashion than it is conventionally used. This is a consequence from the definition of garbage spaces being dependent on an access set. Observe that procedures which place garbage spaces on a free list, do access these garbage spaces. That is:

Fact 4.1.9 Let A be an access set and G the set of garbage spaces relative to A. For any space α in G, there

is an access path b not in A, such that b accesses α.

Marking is the operation of determining all such data spaces which are **not** garbage spaces[Gri5] relative to some access set. **Freeing** is the operation which places all garbage spaces, i.e. not marked spaces, on some free list. We will not enter into greater details with respect to how marking and freeing is performed. Neither will we study the conditions and data structures necessary to implement garbage collection.

```
pointer current_space, next_space; area dynamic_area;
based bit mark;
 ...
current_space:=origin(dynamic_area);
while current_space in dynamic_area do
GF1:    next_space:=current_space+size(current_space->);
        if current_space->mark='0'B
           then free(current_space);
           else current_space->mark:='0'B;
        fi;
        current_space:=next_space;
od;
```

Figure 4.1.4 A simple freeing algorithm.

In figure 4.1.4 we show a simple freeing algorithm. Notice that the statement labelled GF1 performs address computation, i.e. an access refinement within "dynamic_area". Notice that we are assuming size to be computable and, also, that it delivers the extent of the space pointed to by "current_space". The "unary" operator -> is required in order to compute the extent of the space pointed to by

"current_space" rather than the extent of "current_space" itself. The field "mark" contains '0'B if the node has not been marked, and '1'B otherwise. The type attribute based used to declare "mark" indicates that "mark" is to be accessed by means of some pointer. In this example we are not concerned with enforcing type-wise correct accesses.

Defn. 4.1.10 The natural access set NA, is an access set which contains:

i- all language processor defined access functions, including the language processor defined access refinements; and

ii- all user defined access functions which do not use a user defined access refinement.

Notice that all spaces inaccessible to the natural access set define the conventional set of garbage spaces[Knu1]. Consider now figure 4.1.4. Suppose that there is no language processor defined freeing procedure for the space "dynamic_area". That is, if some space in "dynamic_area" is not marked after performing the marking procedure, this space is not accessible by means of access paths in the natural access set. But then such a space cannot possibly be freed. Observe that a situation like this may occur in PL/1 with respect to based variables. That is, if PL/1 pointer values are carelessly changed, unretrievable garbage spaces may build up (pollution, as called by Wegner[Weg2]). Formalizing we have then:

Theorem 4.1.11 There is a computational problem, which can be solved only if the programmer is able to generate names in a computational way.

We did not include into the natural access set all those operating system defined access paths which are not

known to the compiler or interpreter. That is, storage manager access functions have not been included into the access set. This has to be so, since otherwise garbage spaces would never occur with respect to the natural access set.

The example used is not unrealistic. It corresponds exactly to the problem found when attempting to implement languages which make use of garbage collectors, e.g. ALGOL68, PASCAL, LISP1.5, SNOBOL4.

Addressing environment is the set of rules which are used to distinguish one among the several data spaces bound to a same textual name. The addressing environment is usually established by global parameters. If used for this purpose, these parameters are usually hidden from the user. The user is aware of them, though, due to the language description. Some examples of global parameters are:

a- display vectors. These parameters are used in block structured languages to choose one among several activations of the same textual name. Usually the name chosen will be the most recently activated one.

b- activation time. Such a parameter is used to distinguish references to specific instances, as opposed to the most recent instance as in the case of display vectors. For example, PL/1 label variables may be global and, thus, when used may refer to an already terminated block. As suggested by Fenichel[Fen1], problems of this sort can be eliminated by associating an activation time parameter with the label value. Thus, an actual transfer of control is valid only if it transfers to a block bearing the same activation time as that defined within the label

variable used. Furthermore, due to the deallocation rules, any block bearing a more recent activation time can now be deallocated.

c- textual reach. This parameter discerns among several different textual names which possess the same identification, i.e. string.

Global parameters could also be user generated. For example, we could use a static language such as FORTRAN IV to implement a block structured environment. In doing this, some form of display vector would be created, and all accesses within this artificial block structured environment would use this display vector as a global parameter.

Parameters are usually extremely sensitive to unrestricted changes. For example, a change to the effective type field of a union typed data space, represents an error if it occurs externally to the union type descriptor. However, if the change occurs internally, or locally for that matter, to this descriptor, it does not necessarily constitute an error. A less evident example is that of the "stack" type in figure 3.3.1. Externally to this type descriptor, any change to the top pointer "first" would probably constitute an error. Internally to this descriptor, however, we must be free to change the value of "first", since, otherwise, we would be unable to implement the type "stack". We have then:

Fact 4.1.12 Formal parameters used by a type descriptor may be accessed from the exterior only by means of an explicitly defined access function.

Corollary 4.1.13 Formal parameters required by a type descriptor must be received by value, i.e. an

internal copy of the actual parameter's value must be generated.

In many cases the parameters sent to a type descriptor are needed only at compile time. For example, the "user_type" parameter of "stack of user_type" in section 3.3 does not have to be kept at run time. Forcing parameters to be local should, again, not be thought as preventing sharing. Notice that the formal parameters we are refering to, are those which correspond to parameters used to complete a type descriptor.

The existence of internal values would suggest a retention mechanism as defined by Johnston and Berry[Joh2, Ber2]. However, opposed to their concept of retention, our model does not imply garbage collection for all spaces. For instance, if rules similar to ALGOL60 apply to a textual name N bound to a space characterized by the name n, deallocation of this space may occur when n is deactivated with respect to N.

With respect to global parameters, there is nothing to add. This is due to the fact, that we can always build a type descriptor to which a given "global" parameter is local. Thus, we may restrict access to internal data in the same way as described above.

Within the examples shown above, we have only mentioned descriptional parameters. Observe though, that the same applies to positional parameters. Furthermore, incorrect assignments to name typed subspaces frequently cause errors which are extremely difficult to trace[Hoa7].

4.2 Dynamic Space Management.

In this section we will study how spaces are made available to program modules. Although our emphasis is on dynamic space allocation, the discussion can easily be adapted to static space allocation.

Modules or groups of modules, usually obtain spaces, i.e. areas, in large portions. These areas are then partitioned into data spaces by these modules. Such a module, or group of modules, will be called a program level.

Let M be a module providing an area A to some program level L. M will be said to precede program level L with respect to A. Suppose now that M would precede itself with respect to A. Thus M would make A available to itself, where A is unknown to the program level containing M before being made available, contradiction. Formalizing we have:

Lemma 4.2.1 A module M cannot precede itself with respect to some area A.

Using the same symbolism as above, notice that within M the area A is a data space within some, usually larger, area A′ known to M. That is, areas are made known to program levels by partitioning areas known to some preceding module. It follows from lemma 4.2.1 that for every program level L using an area A, there is a module M which precedes L with respect to A. Furthermore, the bare machine precedes all program modules. We may conclude then, that from the point of view of making areas available, it suffices to study the partitioning of areas into data spaces. For example, the module which implements the run time stack of some block structured language, precedes the user written procedures using stack frames of this stack.

Similarly the storage management module preceds these stack implementation routines.

Completing the discussion about areas, we must mention that areas are typed. There are two cases to consider:

i- the structuring of the data spaces within the area is known. In this case the area could receive its exact type already when being requested. For example, in ALTRAN all areas (blocks) hold data spaces which are homogeneously typed.

ii- the structuring of the data spaces within the area is not known. Thus, the type of the area cannot possibly be known when the area is being requested. In this case, the area is supposed to be typed no_type [see section 3.2], thus any access to a subspace of this area must be preceded by a typing operation.

A consequence of theorem 4.1.6 [access to proper subspaces of a simple data space requires access refinements] is that the partitioning of an area into data spaces is possible only by means of an access refinement. Ideally this access refinement would be performed by a language defined construct. This implies the existence of language defined partitioning and compacting primitives.

We will define now a construct which serves as a basis for the implementation of the aforementioned primitives. It should be noted though, that, in some cases, this construct must be provided by the user, thus implying the need for user defined access refinements.

Within each program level i there is a natural access set NA_i. The set of data spaces in an area A_j at this program level i, can be partitioned into two disjoint sets

with respect to the natural access set NA_i. These are the accessible set of A_j and the inaccessible set of A_j, where the inaccessible set contains all data spaces in A_j which are garbage spaces relative to NA_i.

Let us examine now whether the natural access set could be defined in such a way that the inaccessible set of a given area is always empty.

Consider the following construct:

skip <name_value> in <area>

This construct delivers the name of the space adjacent to the space characterized by <name_value>. We will not enter into details with respect to how adjacency is defined. For example, adjacency could be virtual, e.g. column elements of a matrix stored by row. The value returned by skip is null whenever <name_value> or the name which should have been returned by skip characterizes a space not completely contained within the area <area>. Observe that the names <name_value>, as well as <area>, could have been delivered by some expression.

Skip allows the scanning of an area, possibly starting at some "random" origin. This does not seem to be a major restriction, since spaces which are only accessible through skip are garbage spaces in a sense. Thus gaining random access to such spaces does not seem to be very meaningful. There are several examples where constructs such as skip occur naturally. One such example is the processing of a sequential file.

With some minor modifications, skip could be transformed into an area partitioning primitive, say obtain. For example, <name_value> could define the extent of the space

being required. Obtain would then deliver a name satisfying this requirement if possible. After computing the name to be delivered, obtain positions itself at the beginning of the next adjacent free space. Several different mechanisms could be devised in order to implement obtain. Each of these mechanisms would be particularly well suited for some class of applications. This points into the direction that primitives such as obtain are likely to be defined by the user.

We will call space invasion any computation delivering an incorrect name typed value. Notice that space invasion may occur due to the incorrect computation of one or more fields of a name, i.e. medium, address and/or extent. It should also be clear that the skip construct above may cause space invasion. Furthermore, if we want to assure mechanically the sequential multityping of spaces, all language defined constructs must be such that space invasion cannot possibly occur, i.e. that they are space-wise correct.

Lemma 4.2.2 The skip construct assures sequential multityping, iff following conditions hold:

i- the extent of the space α characterized by <name_value> can be determined exactly; and

ii- the extent of the space β characterized by the name returned by skip can be determined exactly; and

iii- accesses to β satisfy the conditions of theorem 3.5.3 [sequential multityping conditions for binding].

Proof. The failure of conditions (i) and/or (ii) implies that space invasion could occur due to the use of

skip, thus, sequential multityping cannot be assured. The failure of condition (iii) contradicts the conditions of theorem 3.5.3, consequently β could not possibly be sequentially multityped.

Suppose now that all conditions hold. Due to the existence of <area> within the skip construct and due to conditions (i) and (ii), it follows immediately that a correct name characterizing β can be computed. Furthermore, due to condition (iii), β is necessarily sequentially multityped.

Suppose now that the conditions of lemma 4.2.2 hold for all data spaces in <area>. Skip could then be used to scan all data spaces in <area> in a type-wise correct fashion. On the other hand, if for some data space α in <area> the conditions of lemma 4.2.2 do not hold, type-wise correct accessing of α can no longer be assured when using skip. Consequently skip cannot be included into the natural access set if this set is supposed to preserve type-wise correctness of accesses. Formalizing we have:

Theorem 4.2.3 The type-wise correct accessing property of the access paths in the natural access set enlarged with skip relative to some <area> is maintained iff the conditions of lemma 4.2.2 hold for all data spaces in <area> accessible through skip.

The conditions of theorem 4.2.3 depend on the type-wise correct binding condition of theorem 3.5.3. In many cases though, we do not know the exact type of a subspace β which name has been delivered by skip. However, we are able to determine the exact extent of β. For example, storage management routines may incorporate extent information into the data spaces delivered. Thus the data

spaces α_i in some area A can be scanned by simply retrieving the value of the extent of a given α_i.

Let α be a data space. Let α_i be a subspace of α which exact type τ_a is unknown, but which extent can be determined exactly. Let $\langle b,\tau_b\rangle$ be the name delivered by skip characterizing the space α_i and implying the type τ_b, where $\tau_a \neq \tau_b$. If any of the following properties are met by $_b$, the conditions of theorem 3.5.3 will still hold:

i- except for the extent enquiry no other access (not even typing accesses) are possible to data spaces of type τ_b. In this case τ_b will be called no_access. It should be clear that theorem 3.5.1, and consequently theorem 3.5.3, is trivially satisfied by no_access. Observe that no_access could be used within the freeing procedure of figure 4.1.4, where the information carrying portion of the node being examined will not be accessed;

ii- when using a τ_b implying access path, α_i may only be copied to another data space β_j, where β_j is also of type τ_a. τ_b will be called filler. Thus, the copy operation is achieved by preceding and succeeding the actual data moving operation with identity conversions in such a way that the original type τ_a is preserved. Hence the conditions of theorem 3.5.3 are met. Notice though, that all access paths bound to the copy, i.e. β_j, must imply the type τ_a. In some cases this cannot be verified easily in a mechanical way. Thus we cannot always rely on the defineability of a type such as filler. Observe that filler is used when performing operations such as storage compactation.

iii- α_i is effectively typed τ_b during, or immediately after the processing of skip. Notice that this is the case when placing the space α_i on some free list.

No other single case can be devised which still satisfies the conditions of theorem 3.5.3.

In section 4.1 we have mentioned the need for access refinements. We may safely assume that access refinements defined by the language processor always generate type-wise correct name values, i.e. target type and object type satisfy theorem 3.5.3. On the other hand, we cannot assume the same to happen with respect to user defined access refinements. We have shown though, that there are cases where user defined access refinements are required. We must find then the conditions which make user defined access refinements indispensable. Once these conditions have been found, we are able to know how much power of expression is lost when forbidding user defined access refinements altogether.

Theorem 4.2.4 Let $\langle b,\tau_b\rangle$ be the type τ_b implying name delivered by skip. Furthermore, let b characterize the data space $\langle\alpha_i,\tau_a\rangle$ of type τ_a. Let A be the area containing α_i. User defined access refinements are necessary iff:

i- the skip construct is not provided by the language; or

ii- skip relative to A fails the condition of theorem 4.2.3■

Proof. If either of the above conditions is true, the skip construct cannot be incorporated into the natural access set. Suppose now that the two conditions are false. Thus the language provides

the skip construct and skip relative to A can be incorporated into the natural access set. Suppose, furthermore, that $\langle b, \tau_b \rangle$ access paths could only be determined by user defined access refinements. Since we are assuming type-wise correct accesses, τ_b must satisfy the conditions of theorem 3.5.3 with respect to τ_a. But the same also happens with respect to skip. Furthermore, for any valid τ_b skip may deliver $\langle b, \tau_b \rangle$ by means of an appropriate type declaration.

Type-wise correctness implies space-wise correctness, thus b must be a space-wise correct name. This implies then, that the structural definition of the area A must be taken into account when computing b. Now if this structural definition can be made available by means of possibly dynamic declarations, there are then access paths in the natural access set which emulate the computation of b without necessarily degrading the efficiency of accesses by any considerable amount. Suppose then, that the structural definition of A cannot be made available. But then it is also not available when computing b by means of a user defined access refinement. Thus the user defined access refinement can compute b only by means of parameters (e.g. offsets) abstracted from a previous computation which delivered b. Furthermore, this computation must use access paths defined within the natural access set, since otherwise space-wise correctness could not possibly be assured. But the same could be achieved by skipping over an appropriately defined

space of type no_access or filler. Thus skip could be used with the same degree of efficiency than user defined access refinements, contradicting thus their necessity.

In proving this theorem, we have frequently made the requirement that declarations be provided. This might be quite annoyant in some cases. On the other hand, preventing the existence of user defined access refinements increases the degree of confidence of the program, since several possible pitfals have been eliminated. Conditions (i) and/or (ii) are usually true when defining complicated storage management modules. That is, the required definition of adjacency has not been foreseen by the programming language designer. We may conclude, however, that the cases where user defined access refinements are necessary are seemingly few.

Besides the cases where user defined acces refinements become necessary, there are also cases where they may aid in increasing the efficiency of a program by a considerable amount. It should be obvious though, that these efficiency considerations presuppose a deep understanding of the program in question, in particular of its bottlenecks. Consequently, requiring the programmer to satisfy some protection pre-requisite before defining an access refinement does not seem too much a burden and, possibly, it prevents the misuse of such definition constructs.

We will assume then, that user defined access refinements are normally forbidden. If it becomes necessary to incorporate a user defined access refinement, protection requirements must first be satisfied. That is, certain language contructs are available, but only within a

protected environment and not for general use. This protection prevents then the misuse of error prone constructs. Under these circumstances, we may assume that, in the normal case, all accesses are type-wise correct, and that this correctness is established in a mechanical way.

This all points into the direction that modules which partition an area into one or more data spaces are in fact access functions defined within one and the same type descriptor. That is, "storage management" is in fact a type.

Let us devote now some attention to the type preserving types. Let $\tau_i(n)$ be a type having following properties:

i- n is an integer value and it defines the extent of the data space of type $\tau_i(n)$;

ii- for any underlying type τ there is a value n such that τ can be identity converted to $\tau_i(n)$;

iii- the only conversion with $\tau_i(n)$ as domain is the identity conversion back to the underlying type τ;

iv- accesses to a data space of type $\tau_i(n)$ are:
  a- forbidden- in this case $\tau_i(n)$ is no access(n);
  b- restricted to copy accesses, i.e. direct transfer from one data space to another- in which case $\tau_i(n)$ is filler(n);

v- the extent n of a data space of type $\tau_i(n)$ can be retrieved.

The extent n must be provided since the underlying type could define data spaces of any extent. Observe though, that the value n could be abstracted, if the extent

required by the underlying type is known. That is, n could be implied in some cases. There are restrictions on the possible expressions used to deliver the value n of such a type preserving type though. Furthermore, the correctness of these expressions must be establisheable in a mechanical way. One simple way which also solves most of the cases, is to define a structure where one of the fields denotes the extent. Clearly this field must be constant once the data space has been typed. For example, consider following construct:

```
type var_size_node(integer extent)=
        struct(integer constant extent copy;
               filler(refer extent) info);
```

where copy denotes that the value of the formal parameter with identical textual name is to be placed in this field. Recall that parameters of type descriptors are passed by value [corollary 4.1.13]. refer denotes that the required value is to be found in the field with identical textual name. Observe that, in this example, "extent" is an intrinsic parameter. Within a storage management type descriptor, data spaces made available to the exterior could be typed "var_size_node". In principle it would be possible then to perform all scanning operations and still assure type-wise correct accesses to data spaces.

4.3 Establishing a Controlled Environment for Name Typed Values.

Even when target typed, name typed values, say pointers, are too general. This follows from the fact that they allow gaining access to any space of the target type. For instance, in ALGOL68 a ref real may access any variable of type real. However, only few of such variables are really supposed to be accessed by this means. In many cases we are also able to determine the logical "life-time" of a data space. However, due to the general availability of pointers, we cannot force a space to be deallocated when its logical life-time has expired. This follows from the fact that an unexpected copy of a pointer could have been made, thus deallocation could result in disaster. In practice it has become evident, that the certification of programs making use of pointers is quite difficult, by whatever means one chooses. This all suggests that the availability of pointers should be restricted as much as possible[Hoa7].

The most drastic restriction, would be to forbid the existence of pointers altogether. However, this restriction may cause problems of a different nature when attempting to circumvent the inexistence of pointers. Another restriction which is not as drastic, would be to forbid pointers to internal data spaces to be transmitted to the exterior of a program module, e.g. type descriptor. This would effectively shield a user of a given program module from knowing of the existence of pointers. A step in this direction are the access typed values (functions) which we have introduced in section 3.3. Such a value is in fact a pointer, however, the user sees it only as a variable of the corresponding target type, e.g. ALTRAN data values.

Access functions deliver access typed values, i.e. names. These values may then be assigned to an access typed space, i.e. an access variable, or may be used immediately to access the required data space. Except for the assignment of the result of an access function elaboration and parameter association, there may not be any other access typed value assignment. We cannot forbid the existence of access variables altogether, since a given access function could contain internal side effects and, consequently, should be elaborated only once for a group of one or more effective accesses. For example, an access function could deliver a new record of a sequential file for each elaboration of this function.

We must stress the fact that access values are not values on their own. For instance, let A be an access variable with target type T. If A is used in some expression, A stands for a value of type T rather than for a ref T value. Moreover, If A occurs in an assignment, A stands for the space which will receive a value of type T rather than for a space of type ref T. Thus, we may think of A as being a "simple" variable of type T. This suggests then a construct which, besides assigning a value to an access variable, also defines a scope within which this value is defined. For instance:

with <id>=<access_function> do <statement_list> od;

where <access_function> will be elaborated and the value it returns will be assigned to the access variable <id>. Within <statement_list> any access to the data space in question will be denoted by <id>. With constructs may be nested and possibly refer to the same <access_function>. Observe that <id> is defined only within <statement_list>. Furthermore, <id> has not necessarily to be declared, since

its target type could be abstracted from the access function used to generate its value. However, if <id> is explicitly declared, the type must be <access_form> <target_type>, where <access_form> is any of access, fetch or store. In figure 4.3.1 we show a concrete example of the use of the with construct.

Constructs like the with construct have been defined both in PASCAL (with) and in SIMULA67 (inspect). The SIMULA67 construct performs an object identification test, as well as providing the with construct. In both cases only one access typed value of a given target type may be present. This follows from the fact that PASCAL, as well as SIMULA67, do not explicitly provide <id> within the with construct. That is, within <statement_list>, record field names are used directly without prefixing it with <id>. Consequently there would be an ambiguity if equal type delivering with constructs would be nested. Qualifying accesses to fields of a structure with <id> may be annoying in some cases, but such a qualification leads to a clearer text within <statement_list> and, also, it prevents the aforementioned ambiguity to occur.

Suppose now that a given program module makes internal data spaces available to the exterior. This could be achieved, for instance, by transmitting a pointer, i.e. the address of such an internal space. We have already mentioned that this would be undesirable. Let us examine then if there is an alternate way to transmit references to internal data spaces avoiding the generality of pointers.

Assume that within program modules some sort of identification is associated with each internal data space, such that each identification corresponds to at most one

data space. Furthermore, assume that a mapping is defined, which maps a given identification onto the corresponding data space if any. Thus, if this mapping is defined as an access function, it follows immediately that it suffices to transmit data space identifications rather than pointers and still be able to access internal data spaces. Formalizing we have:

Theorem 4.3.1 Any computation requiring pointers to internal data spaces to be transmitted to the exterior of program modules, can effectively be transformed into a computation requiring only access typed values characterizing these internal data spaces to be transmitted.

Examples of such identifications are: integers; character strings etc. Examples of the corresponding access functions are: indexing into an array of pointers; hashing functions etc. In extreme cases the identification could actually be a pointer and the corresponding access function would be the identity function.

Notice that data space identifications are not access typed values. That is, when effectively accessing an internal data space, only access typed values are used. Externally however, this data space is identified by a value which is not an access typed value. Furthermore, this identification itself does not stand for an access path, even if it corresponds to a "disguised" pointer.

Defn. 4.3.2 Let M be a program module making internal data spaces available to the exterior. M is said to exist in a controlled environment, if:

i- all external accesses to internal data spaces of M are gained by means of some access function defined by M; and

ii- these access functions may be of any complexity and must successfully elaborate for any access request.

From (ii) we have that access functions could perform access validations such as: controlling identity of users; assuring that accesses are "life-time" correct. This definition shows also, that data spaces which are internal to different modules are kept disjoint whenever program modules exist within a controlled environment. That is, in order to access some data space, the program module instance containing it must be known.

We will expect that the access functions can be elaborated in a deterministic way. This does not mean that access may not be denied. It means, however, that if access is to be gained, all information needed to elaborate the access function must be available. It should be clear that the information required could be obtained by:

i- including additional information into data space identifications;

ii- request information from the module containing the access function, e.g. global parameters;

iii- request information from the module requesting the service, e.g. module identification;

iv- request information from the exterior, i.e. input operations.

Suppose now that data space identifications could be computed externally to the program module. Due to the nature of data space identifications we could conclude that accesses will always be space-wise correct, i.e. no space invasion ever occurs. However, no longer are we able to provide a controlled environment. This follows from the fact that external changes to subspaces of a data space identification may restrict the class of tests the corresponding access function may perform. That is, allowing external changes to data space identifications, restricts the complexity of the associated access function. For example, the validation of accesses by means of passwords contained within the data space identification cannot be performed if these passwords are allowed to be changed externally. We may conclude then, that, in order to establish a controlled environment, data space identifications may not be obtained by computational means externally to the corresponding program module. We may now formalize the properties a data space identification must possess:

Defn. 4.3.3 Let α be a data space which is internal to some program module M. A data space identification of α is a value V of some type τ_1, such that:

i- α and only α is accessed when presenting V to any of the associated access functions;

ii- the associated access function fails for every value of type τ_1 which does not properly identify an internal space;

iii- externally to M, values of type τ_1 are indivisible and cannot be obtained by computational means;

iv- externally to M, values of type τ_1 can access

the data space they identify only by means of some associated access function F.

Theorem 4.3.4 A program module M exists within a controlled environment and theorem 4.3.1 is satisfied, iff M makes internal spaces available by means of data space identifications only.

Proof. From the previous discussion it follows that the condition (iii) of data space identifications [4.3.3] is necessary and sufficient in order to allow access functions to be of any complexity. Thus condition (iii) satisfies condition (ii) of the controlled environment definition [4.3.2]. Since condition (iv) of the data space identification definition [4.3.3] is identical to condition (i) of the controlled environment definition [4.3.2], if follows that conditions (iii) and (iv) are necessary and sufficient in order to establish a controlled environment for the module M.

If conditions (i) and (ii) are met, theorem 4.3.1 holds, since there is an one to one correspondence between pointers and data space identifications. If condition (i) fails, a data space identification would identify more than one data spaces, thus not being in an one to one correspondence with the pointers. Similarly, if condition (ii) fails, access to a "non-existing" space could be gained, thus there would be no one to one correspondence between data space identifications and null pointers.

Notice that neither pointers nor integers satisfy definition 4.3.3. This follows from their respective

violations of conditions (iv) and (iii). Later in this section we will show how values of such types could be qualified in order to satisfy all of the properties metioned above. Observe that, by theorem 4.3.4, if pointers abide to the restrictions of definition 4.3.3, the associated access function could be the identity function, and still a controlled environment for these "pointers" would be maintained. We may conclude then, that the following restriction could be enforced without degrading efficiency or decreasing the language's power of expression.

Restriction 4.3.5 External accesses to some internal data space of some module M, may be gained only by means of access functions defined within M.

There are several interesting properties with respect to data space identifications. For example, property (ii) shows that we could associate control information both with the data space in question, as well as with the corresponding data space identification. This then allows preventing accesses where the control information of both data space and its identification do not agree. For example, we could define a type "forest". Within this "forest" several different "trees" could be made available. If data space identifications contain also some information linked uniquely to each "tree", we can effectively prevent accesses from jumping "trees", even when "nodes" are moved from one "tree" to another.

The control information could also be a function of the number of copies of a given data space identification, e.g. reference counts. How such an information is kept up to date will be studied in section 4.4.

The preceding discussion shows that data space identifications might be quite complex. It shows also, that internal type definitions of some program module must be made available to the exterior. This does not represent a major notational problem, since the type definition could be made available to the exterior in the same way as functions are, i.e. by including them into the interface information of the program module. It constitutes a compilation problem though. For instance, let A be a type descriptor requiring an externally defined type B, and making available an internal type C. If the type B is such that it requires type C to completely compile, it follows immediately that A and B can be compiled completely only when the interface information of both A and B is present. That is, restrictions are imposed on independent compilations, i.e. there are cases where independent compilation cannot be completed at a given compile step. This points into the direction that there might be several compilation steps in order to completely compile some given program.

When defining data space identifications, it was mentioned that, externally to the program module defining them, data space identifications should not be partially changed nor obtained by means of some computation. Let us examine now whether we could enforce this in a mechanical way by means of type checking.

Defn. 4.3.6 Let τ be some type. A type qualification is an operation which transforms the underlying type τ into a type τ' having following properties:

i- there are no subtypes to τ';

ii- the identity conversions $I_{\tau'\tau}$ and $I_{\tau\tau'}$ are defined between τ and τ'.

Defn. 4.3.7 Let τ' be some qualified type. A τ' owning module is a program module which knows both τ' and the underlying type τ of τ'.

In section 3.2 we have introduced the type no_type. For any type τ, a conversion from no_type to τ yields the undefined value. This is particularly important when a data space α of type τ is being allocated, since during the allocation process of α, α will be typed no_type.

Theorem 4.3.8 Let τ' be a qualification of some underlying type . Let M be a program module which uses τ' but which is not a τ' owning module, i.e. M ignores τ. The type-wise correctness of accesses is maintained by M in the general case, iff any conversion with τ' as range has a qualification of τ' or no_type as domain.

Proof. Suppose that C has no_type as domain. Thus the data space α containing the result of C will effectively contain the value undefined. Hence any read access to α must be preceded by a write access as required by theorem 3.5.1. Now if C has a qualification τ'' of τ' as domain, τ' underlies $''$, consequently their type set is identical and the set of valid operations of τ'' is a subset of that of τ'.

Suppose now that the domain τ'' of C is some type other than a qualification of τ' or no_type. Since τ'' is not no_type, the conversion C delivers a value which is read accessible. Now, due to the existence of the identity conversion $I_{\tau''\tau'}$, C also converts τ'' to τ'. But then C must know τ in order to correctly convert, contradiction.

Theorem 4.3.8 shows that type qualifications cannot be used directly in order to implement data space identifications. This follows from the necessity of knowing the underlying type in order to assure type-wise correctness of accesses.

Defn. 4.3.9 A proper type qualification of an underlying type τ, is an operation which transforms τ into a type τ' such that:

i- τ' is a qualification of τ; and

ii- within a module using τ' and not being τ' owning, any conversion with τ' as range has no type or a proper type qualification τ'' of τ' as domain.

The next theorem shows that proper type qualifications satisfy the external requisites, i.e. condition (iii), of a data space identification.

Theorem 4.3.10 Let τ' be a proper type qualification of some type τ. Within a module M not being a τ' owning module, values of type τ' are indivisible and cannot be obtained by computational means.

Proof. That values of type τ' are indivisible follows from the inexistence of subtypes to a qualified type. Furthermore, due to the conversion restriction of proper type qualifications, values of type τ' cannot be obtained by computational means within M.

Now, if the underlying type τ is not known, it follows that we cannot use τ' directly as a name typed value, even if τ is a pointer. Thus any data space identified by a value V of type τ' can only be accessed by means of an

access function accepting V as a parameter. Furthermore, this access function must be defined within a τ' owning module. From this and from theorem 4.3.10, we may conclude that restriction 4.3.5 is indeed enforceable in a mechanical way. For this it suffices to prevent the underlying type of a proper type qualification to be part of the interface information. We are able then to transform types such as integers, strings, pointers, structures, into data space identifications.

Due to the permission of having τ' in the domain of a conversion, it becomes possible then to define relations with respect to values of a qualified type τ', without that this relation has to be implemented by some τ' owning module. As has been pointed out by Knuth[Knu3], this allows the design of more "efficient" programs in some cases. For example, if it is known that the relative position of dynamic data spaces is maintained during all storage administration operations, some lists could be ordered according to the binary value of pointers, allowing thus a faster searching opperation.

Proper type qualifications will be denoted by the keyword shield. There are some implementational problems with respect to shielded types. For instance, data spaces of a shielded type could be of varying extent. Notice that this could be overcome by means of an appropriate extent enquiry. We will not enter into details with regard to these implementational difficulties though.

In figure 4.3.1 we show an example of how the type shield could be used. We show there how a linked list implementation of a binary tree could be provided, so that only data space identifications are transmitted. Observe

```
type tree .(type user_type) =
   begin tree;
      type node=struct(user_type info;
                integer node_id; ref node left, right);
      type reference=struct(ref node pointer; integer id);
      outside_scope tree_ops;
         type node_ref=shield reference;
               ...
         user_type access function get(node_ref what) =
            if what.id=what.pointer->node_id
               then what.pointer->info;
               else fail;
                ...
   end tree;
```

Example of use:

```
   tree a(integer);
   type pointer=a.node_ref;
   with node=a.get(pointer) do ... node:=10; ... od;
```

Figure 4.3.1 Example of the use of type shield.

that "reference" is the underlying type and that it is able to directly access the required node. "node_ref" is the shield (qualification) of "reference". In order to be able to communicate with the exterior, "node_ref" must be available in the interface information, i.e. within "outside_scope".

By means of the fields "node_id" and "id" we can establish following protection mechanism. The value of "node_id" could be modified every time the node is transmitted to or obtained from a free list. Thus, by means of the access validation check shown within the access

function, we are able to verify if a correct node is being accessed. Since the user is unable to modify the identification, we are effectively enforcing accesses to be "life-time" correct.

Observe also that we are using the textual name "what" as if it implies the type "reference". This is valid since, within a τ' owning module, the types τ' and its underlying type τ are synonymous due to the existence of the identity conversions. Thus, if a textual name implies τ' it also implies τ witin a τ' owning module.

When allowing values of any complexity to become data space identifications, the null data space identification is no longer clearly defined in all cases. This shows then, that a function must be defined which produces null data space identifications. Furthermore, this function will have to be user defined in some cases. Since, the language processor assumes a predefined operation which delivers such null values whenever null is used within a program, we conclude that we may have to replace language processor operations by user defined functions. How this can be achieved will be our object of study in the next section.

4.4 Access Monitoring.

In section 4.3 we have shown how to establish a controlled environment for name typed values. For this purpose we have introduced data space identifications. Still these data space identifications are some sort of "pointer", regardless of how much information they actually carry. This follows from the fact that data space identifications could

be copied externally and, thus, we are unable to control the dissemination of name typed information.

When transmitting data space identifications we may want to monitor the use of such values. For example, we may desire to establish reference counts, or prevent copies from being made. In this section we will study how this monitoring could be established. We will not limit ourselves to consider only data space identifications though. That is, we will study how to establish monitoring as such, regardless of the purpose of the underlying type.

In order to reduce the writing effort, language processors define global types, e.g. integer, as well as some global operations which, in a sense, are "type less", e.g. store or deallocate. When monitoring accesses, we may want to substitute some or all of these operators by user defined functions. Notice that some languages allow such replacements for some of the defined operations, e.g. SNOBOL4 "OPSYN" function, ALGOL68 op declaration.

Defn. 4.4.1 Let O be a language processor defined operation. A type T is said to be O-sensitive, if the operation O on values of type T can only be performed by an O-emulating function F defined by T's type descriptor T.

Observe that such a "language processor" could have been defined by the user by means of another language processor. We have then in fact a hierarchy of languages. For each of these languages we must define those operations which could be replaced and, also, the restrictions (protection requirements) which govern this replacement. It follows from this discussion that the operation to be

replaced by an O-emulating function might well be user defined.

According to section 3.4 the type of each parameter must be present in the interface information. Furthermore, when combining modules, the types of corresponding parameters must satisfy lemma 3.5.3 [sequentially multityped binding conditions] as we shall show in chapter 5. Since the type descriptor defines also all O-emulating functions, it follows that we can reduce the problem of replacing an operator by the corresponding emulating function to a type checking problem.

With respect to defining O-emulating functions, two main difficulties arise immediately:

i- the replacement may cause language processor assured type-wise correctness to be lost. For example, the definition of extent enquiry emulating functions may cause space invasion;

ii- some operations are such that the emulating funtion must either report an error or eventually perform the operation being emulated. For example, emulating the deallocation operation must eventually perform this deallocation, otherwise disastrous accesses could occur.

From (ii) it follows that an O-emulating function F may have to refer to the operation O itself. On the other hand, O and F are identified by the same textual name in order to allow the replacement to be performed when matching the interface information. We must then adopt following convention in order to avoid improper replacements:

Convention 4.4.2 Let TX be the textual name of some operation O and also of its O-emulating function F. Within the body of F, any occurrence of TX refers to O and is not replaced by F.

An O-emulating function F will be said to be exact O-emulating, if, when elaborating F, F either reports an error or the operation O is performed with exactly the same parameters as those passed to F. It should be clear, that an exact O-emulating function maintains the the type-wise correctness of the language processor. This follows from the fact that the original operation O is necessarily type-wise correct whenever the language processor assures sequential multityping. Furthermore, this operation is eventually performed whenever F completes elaboration without reporting an error.

It should be clear that many O-emulating functions are quite simple. Thus, if some textual discipline is enforced, e.g. "goto-less" programming, these functions could be shown to be exact in a mechanical way, without increasing the compiling cost by much. Now, in order to avoid misuse of the programming language, we will also here require that some protection convention be satisfied whenever defining an O-emulating function which cannot be mechanically proven of being exact.

In section 4.3 we have mentioned that a dynamic storage allocation module DSA is in fact a type. Furthermore, such a module makes internal spaces available to the exterior. Suppose we could access such a space α_i by means other than data space identifications and still abide to restriction 4.3.5 [internal access via access functions only]. We would have then to define an access function A

which accesses α_i. This access function A may not require any external parameter which is necessarily in an one to one correspondence with the data spaces α_i it can access, otherwise this parameter would be a data space identification. Furthermore, within DSA there is no relationship between the data spaces made available to the exterior, consequently the function A could not be designed in such a way as to provide access to successive data spaces, since this succession is not known. That is, A cannot be a generator function as will be defined in section 4.5. It follows then, that A cannot decide which α_i is to be accessed. There must then be an access function A_i for each α_i, where the name of A_i is transmitted to the exterior. But then this name is the data space identification of α_i. Formalizing we have:

Theorem 4.4.3 There is a computational problem which requires the transmission of data space identifications to the exterior.

A consequence of this theorem is that there might be cases where access to data space identifications must be monitored.

It should be clear that operations such as "deallocation" of spaces are internal to the "dynamic storage" type descriptor. Suppose now that the deallocation is governed by reference counts, thus all operations which might modify the contents of a data space containing a reference (data space identification) must be monitored. In particular the store operation must be monitored. Since store is usually a globally defined operation, it follows that a store emulating function is required here.

```
type node = struct(integer count; user_type info);
type reference =
   begin reference;
      ref node pointer;
      outside_scope ops;
         emulate store(reference value) =
            begin store;
               reference save_pointer = pointer;
               store(value);       /* changes "pointer" */
               if pointer≠null
                  then pointer->count:=pointer->count+1;
               fi;
               if save_pointer≠null
                  then save_pointer->count:=save_pointer->count-1;
               fi;
               if save_pointer->count=0
                  then user_type.nullify_pointers( save_pointer );
                       deallocate( save_pointer );
               fi;
            end store;
      end ops;
   end reference;
```

Figure 4.4.1 Definition of a name typed value establishing reference counts.

In figure 4.4.1 we show how reference counts could be maintained by means of defining a store emulating function. Notice that the type descriptor of "reference" is internal to some type descriptor, say "dynamic storage". Notice, furthermore, that "pointer" is the receiving data space when store is performed.

It follows from our type equality definition 3.4.4 in section 3.4, that all modules which interchange values of type "reference" must refer to one and the same definition of this type. Consequently all store operations can be effectively replaced by the corresponding store emulating function, since this emulating function is part of the "reference" type descriptor.

The example shown in figure 4.4.1 is not complete. For instance, operations such as deallocation must also be monitored. We have omitted these functions since it would not contribute to the understanding of the emulation function definition shown there. Notice that the store emulating function shown in figure 4.4.1 is an exact emulating function.

When using reference counts, a data space α is deallocated whenever its corresponding reference count I reaches the value zero. It follows immediately that the reference count J of the data space β must be decreased if α contains a reference to β. Now, for reasons of transparency, we do not want to know the "user_type" of the data space α being deallocated. It follows then, that "user_type" must provide the means to set the references (data space identifications) contained within α to null. The easiest way is to define within "user_type" a function "nullify_pointers" which stores null in all name typed fields of the data space being deallocated. These store operations will then be monitored again. Thus, by means of recursion, eventually all data spaces to be deallocated will be effectively deallocated. Observe that, by means of "nullify_pointers", a given data space could provoke the updating of several "dynamic storages". It follows then, that O-emulating functions must be implemented in such a way that they allow

recursion, i.e. multiple instances. This could be achieved, for example, by means of "safe routines" as in IPL V[New5].

The operation "nullify_pointers" should be a language processor defined operation. Notice though, that the underlying structure must be known in order to implement this operation. Thus, it depends on the declaration of "user_type" and, consequently, its definition is completed within "user_type". Notice also that "user_type" may stand for a hierarchy of types. Thus, "nullify_pointers" might be partitioned into several sub-operations. This emphasizes then, that "nullify_pointers" be a language processor defined operation.

We have mentioned in section 4.3 that null is in fact a function wich is defined for name types, e.g. pointers, and for data space identification types. It follows then, that "nullify_pointers" needs only to modify those fields of the data space α being deallocated which currently bear a type defining the operation null. Thus fields containing only computational values need not be changed, reducing thus the deallocation run time cost.

4.5 Generator Functions.

In this section we will study a class of access functions which are capable of accessing internal spaces of some module (usually a type descriptor), without requiring data space identifications as external parameters. Functions of this kind will be called *generator functions*. We will not restrict generator functions to be just access functions though. That is, a function which computes the values of successive elements of some set will also be

considered as being a generator function. For example, a random number generator will be considered as being a generator function, although it is usually not implemented as such.

Generator functions are particularly interesting for the fact that they allow to define, once and for all, the form in which consecutive elements of an (ordered) set are to be generated†. Thus the task of writing and certifying programs may be reduced considerably, since this has to be done only once for several effective uses of a given algorithm, i.e. generator function.

Usually generator functions are defined internally to some module, e.g. type descriptor. This has the advantage of keeping together in one module the set of valid operations on internal data spaces. Thus generator functions increase the structuring of programs as well as the interchangeability of modules. Due to being internal to other modules, generator functions know both the "structure" of the underlying set and also the order in which the elements of this set are to be delivered. This allows then the design of efficient generator functions without having to destroy the transparency of module implementation.

Notice that the "sets" we are refering to are in fact sequences, that is, they could contain repeated elements. For example, a random number generator could deliver several times the same value, possibly even a contiguous sequence of equal values. However, we assume that every value delivered by this random number generator is a different element of some set.

---

† Recall that a tree traversal algorithm defines the ordering of the (sub) set of nodes it traverses.

Generator functions usually possess internal parameters which must be kept from activation to activation. By means of these parameters the current (or next) element to be accessed is determined. Furthermore, these internal parameters are normally updated for every activation of the generator function. That is, generator functions usually produce internal side-effects and, consequently, may deliver different values for successive activations. Finally, in order to produce the current (or next) element of the set to be processed, the generator function may have to resume elaboration where it last went off. We may conclude then, that generator functions tend to be coroutines. We will study the problems relative to coroutines later in this section.

Generator functions are further distinguished from conventional functions in that they usually define following three entry points:

i- initialization entry- which prepares the generator function to deliver the first element of the set, or actually delivers it;

ii- successor entry- which advances the element "cursor" by one element. That is, by repeatedly activating the generator function through the successor entry, we are able to effectively scan the elements in the ordered set;

iii- termination entry- this is a predicate which determines whether all elements have already been examined.

Notice that these three entry points do not necessarily have to be provided. For example, a random number

generator could be initialized whenever an instance of it is created. Furthermore, termination predicates are usually absent from random number generators. Thus, a random number generator may be implemented as a conventional function, although it is considered to be a generator function.

From what has been said so far we may conclude:

Fact 4.5.1 Generator functions are modules possessing their own storage requirements and making one or more manipulative operations available to the exterior.

Observe that type descriptors have the same properties. Thus the mechanisms developed for type descriptors could also apply to generator functions. Since types and generator functions exist for a different purpose, we will distinguish between them by using respectively the keywords type and generator.

```
set S(some_type);
generate A=S.elem_gen.first by S.elem_gen.successor
                    until S.elem_gen.last do;
   generate B=S.elem_gen.first by S.elem_gen.successor
                       until S.elem_gen.last do;
      output A, B;
   od;
od;
```

Figure 4.5.1 Ordered pair generator. First version.

Consider now following problem. Given some set S, produce a listing containing all ordered pairs of elements in S. This could be achieved by a program similar to that of figure 4.5.1. We are not concerned here with what these sets represent, e.g. data base records bearing a given

property, nodes of a tree or a graph. What we want to point out though, is that successive elements of such sets cannot always be obtained by simple addition, e.g. indexing.

The construct:

generate <control_var>=<origin> by <successor>
until <termination> do ... od

is similar to the ALGOL60 for statement. It is used with respect to generator functions though. Thus, <origin>, <successor> and <termination> are, respectively, the initialization, successor and termination entries of the generator function. <control_var> is usually an access typed variable and refers to the element of the set which is currently being processed.

Within the program of figure 4.5.1, "set" is a type which, besides storing elements of some set, "set" also makes the generator function "elem_gen" available to the exterior. This generator function scans all elements in the set "S". "elem_gen" provides following entry points:

i- "first" which resets "elem_gen" and provides an access typed value refering to the first element in the set;

ii- "successor" which advances access to the next element in the set;

iii- "last" is a predicate which returns true iff all elements of the set have been examined.

Suppose now that there is only one instance of the generator function "elem_gen". When the internal loop, i.e. "generate B=...", terminates, the generator function instance is necessarily in a state where the the until test

yields true. Now, when terminating the internal loop, the external loop is resumed, i.e. the next left hand element of the ordered pair is generated. However, there is only one instance of the generator function, it follows then immediately that the external loop is terminated since this unique generator function instance reached the termination condition. It follows then, that the program shown would be in error, since only those ordered pairs are listed for which the left hand element is the first element of the set being traversed. Formalizing we have:

Fact 4.5.2 There may be several instances of a generator function, each at a different stage of elaboration and each possibly related to the same data space or module instance.

Observe that even when the generator function is a simple addition operation the result above is true. In this case, the multiple instances are usually embedded into the program's code, e.g. by multiple expansions of the loop "macro".

We have already mentioned that generator functions and type descriptors are similar from the implementation point of view. Thus, instances of generator functions could be created in the same way as data spaces of a given type are created, i.e. by means of a declaration. We will assume then, that all generator function instances are declared. Obviously, we could define language constructs which would provide such declarations in an implicit way. Generator function instances could also be created explicitly by means of some operator, e.g. the new(<class>) operator of SIMULA67. However, we will not study such constructs in this dissertation.

```
set S(some_type);
S.elem_gen outer, inner;
generate A=outer.first by outer.successor until outer.last do;
   generate B=inner.first by inner.successor until inner.last do;
      output A, B;
   od;
od;
```

Figure 4.5.2 Ordered pair generator. Second version.

In figure 4.5.2 we show an example of generator function instance declarations. Since "outer" and "inner" are different instances of the generator function "elem_gen", it follows immediately that there is no interference between the two generate statements in figure 4.5.2. Observe also that "outer" and "inner" are defined relative to the same data space "S" of type "set".

In figure 4.5.3 we show the type descriptor "set". As mentioned earlier, the type "set" makes available the generator function "elem_gen". The example shown should be clear, since it reflects exactly all what has been said about generator functions so far.

A generator function instance declaration causes the creation of a generator function instance whenever control passes through the program section containing, or elaborating, the declaration. Thus, the internal spaces of a generator function, i.e. the activation record, are allocated and remain allocated even if none of the operations defined by the generator function are actually being elaborated. It follows then, that generator functions establish an environment within which coroutine instances may be kept from activation to activation without that this

```
type set .(type user_type) = begin set;
   type element=struct(ref element next; user_type info);
   ref element head:=null;
   outside scope set_operations;
      generator elem_gen = begin element_generator;
         ref element current:=head;
         boolean end_reached:=head=null;
         outside scope entry_points;
            user_type access function first =
               if end_reached then null; else head->info; fi;
            user_type access function successor =
               if end_reached then null;
                  else head:=head->next;
                       end_reached:=head=null;
                       if end_reached then null;
                          else head->info;
                       fi;
               fi;
            boolean function last=end_reached;
         end entry_points;
      end element_generator;
   ...
end set;
```

Figure 4.5.3 Definition of the type "set".

has to be known externally. Furthermore, these internal spaces may be dynamic in the sense that space allocation may be delayed until the space is actually needed, and deallocation may occur as soon as the generator function instance is no longer needed.

Observe that there are several creation steps associated with generator functions. One step is the creation of

the code sections and occurs usually at compile (load) time. A second creation step occurs when a generator function instance is created. A third creation step occurs when an internal function of the generator function is started to be elaborated. Observe that the activation records of the internal functions are linked to the "activation record" of the generator function instance and, consequently, may be kept as long as the generator function instance is kept.

Let us consider now a generator function which performs the infix traversal of a binary tree (post-order traversal in Knuth's terminology[Knu1]). Whenever a node to be visited is found, the infix generator function must relinquish control to the calling procedure. Furthermore, there is a "past history", e.g. a stack, associated with the traversal algorithm. This past history must be preserved from activation to activation. Thus, the generator function must be implemented as a coroutine. Now, the past history could be maintained in an implicit way by means of a recursive procedure. We may conclude then, that this infix generator function could be implemented as a recursive coroutine.

Notice that recursion could be avoided if coroutines themselves keep track of the past history. This would be the case in languages such as SLIP[Wei2,Smi2] which do not provide the ability to explicitly define recursive functions.

Summing up we have:

a- several instances of a generator function may exist at the same time. Each of these instances being

independent from the others in the sense that there is no recursion relation between them;

b- generator function instances are created by means of declarations. The internal function instances, possibly coroutine instances, are linked to this generator function instance;

c- the internal functions which actually perform the computations of the generator function tend to be coroutines, possibly even recursive coroutines.

We will examine now the problems relative to implementing coroutines. A module may receive control only through well defined entry points. There may be several entry points to one module. Every entry point defines an effective entry value which determines where elaboration has to begin (or resume) when control is transfered through this particular entry point. Effective entry values may be variable. For example, a coroutine usually allows elaboration to resume at one of several predefined points within the text. Whenever a coroutine deactivates, it sets the corresponding effective entry to refer to one of the elaboration resumption points. We need thus a deactivation construct which also sets the value of the effective entry to refer to the resumption point.

In order to pass control back to the caller, we need an effective return value. This value is also associated with the entry point through which control has been passed to the module. Thus, an entry point could be viewed as the name of a composite data space containing following kind of values <effective entry; effective return>. It should be noted here that an effective return value could be a more complicated structure than just a label, e.g. address. For

instance, an effective return value could determine the actual return point, as well as the entry points of several error recovery entries. In some cases the data space containing the effective returns may be shared by several entry points. This is the case, for example, in subroutines (or functions) which posses multiple entry points, e.g. the sine/cosine function.

A module instance is an "elaborateable" copy of a program module. Usually a module instance consists of a (shared) portion of code and a (non-shared) portion of working storage, i.e. an activation record. Module instances are created when a creation section associated with the module is elaborated. Usually there is only one creation section per module and this creation section is provided in an implicit form by the language processor. Observe that, when a module is a macro, the creation of a module instance corresponds to a macro expansion.

A creation section which is implied by the language processor is called a prologue and its elaboration preceds the elaboration of the first statement of the module. In this case the textual name of the entry point is conventionally also the textual name of the module containing this entry point. Thus, the effective entry value can easily be initialized to refer to the appropriate prologue. We will allow then entry points and the corresponding modules to possess the same textual name.

Entry points which effective entries refer to a creation section will be called creation entries. Observe that a creation entry is transformed into an activation entry whenever the corresponding effective entry value is

set to refer to a program section which is not a creation section.

In some cases there are several entry points all refering to the same prologue. This occurs for instance, when defining multiple entry point functions (modules) such as the sine/cosine function. This shows that there may be parameters by means of which the flow of control is directed after the creation has been completed. These parameters are usually provided in an implicit form by the entry point effectively used, e.g. "sine" or "cosine".

Creation may occur all at once, e.g. as in FORTRAN IV, or it could occur in several steps, e.g. in ALGOL60 the shared portion is created at compile (load) time, whereas activation records are created and destroyed during execution. Termination is the operation of destroying a module instance. It should be clear that, if a retention mechanism is implemented[Ber2,Ber4,Weg2,Joh2], actual termination could be delayed with respect to the instant when the termination operation is performed.

Consider now recursive coroutines. With respect to deactivations, a recursive coroutine may deactivate and return control to the exterior, or it may deactivate (terminate) and resume elaboration of another instance of this coroutine. This implies then the existence of at least two effective return values and, also, of two entry points. It implies furthermore, the necessity of deciding which effective return is to be used when deactivating. Since this decision depends on the algorithm to be implemented, it must be provided by the programmer. It follows then, that deactivations must be able to name the entry point containing the return point to be used.

Allowing a textual name to stand both for a module name and an entry point name may cause some unusual control flow interactions between modules. This follows from the fact that, in some cases, the effective return value to be used is not the one associated with the entry point which textual name is equal to the textual name of the module within which the deactivation operation occurs. That is, some deactivations could be relative to an entry point associated with another module. Not only this, the effective entry value of some entry point may be set to refer to a statement in a module other than the module bearing the same textual name. This occurs for example, with respect to recursive coroutines. We will not examine here whether the syntactical nesting of such interacting modules is feasible in the general case, nor will we examine whether it presents a more structured solution.

A module instance may only be activated if it has already been created and has not yet been terminated. Thus, in order to activate a non existing module instance, first some creation section of this module must be elaborated. In order to prevent incorrect execution, the effective entry values must be initialized to refer to either a creation section or to abort. Observe that if control is passed to abort, an error condition is flagged and the program execution may reach a premature end. By a similar argument we may conclude that all effective return values must be initialized to abort.

From what has been said above, it follows that a coroutine instance can be created by the first transfer of control to the coroutine. This can be achieved by means of an appropriate initialization of the effective entry values associated with the entry points of the coroutine. Fur-

thermore, subsequent activations of this coroutine can be prevented from creating new instances, simply by setting the effective entry values to refer to some portion of the code which is not a creation section. It follows then, that coroutine instances need not be created externally to the module defining the coroutine.

Observe though, that there might be cases where we want coroutines to be created previous to the first activation. There are then at least two entries, a creation entry and an activation entry. Since, as noted above, only existing instances may be activated, it follows that the activation entry names must be qualified with the instance name whenever control is passed to it. It should be clear that the creation and the activation entry names could be provided implicitly by the language processor by means of some operators, e.g. create and activate.

In figure 4.5.4 we show a partial definition of the type "binary_tree". Within "binary_tree" we define the infix traversal generator function "infix". The purpose and implementation of this generator function has been described earlier in this section.

Within "infix" two procedures are defined. The recursive procedure (function) "start_subtree" which performs the actual traversal, and the coroutine (co_function) "next" which serves as a communications link with the exterior of "infix". That is, control can be passed to "infix" only through "next". It follows then, that "start_subtree" must deactivate through "next" whenever a node to be visited has been found. Observe that deactivation through "next" requires also the passing of an access typed value characterizing the node to be visited.

```
type binary_tree .(type user_type) = begin binary_tree;
   type node=struct(ref node left, right; user_type info);
   ref node root:=null;
   outside_scope operations;
      generator infix = begin infix;
         function start_subtree(value ref node pointer) =
         begin subtree;
            if pointer≠null
               then call start_subtree(pointer->left);
A:                  deactivate next(pointer->info);
B:                  call start_subtree(pointer->right);
            fi;
         end subtree;
         outside_scope only_known_entry;
            user_type access co_function next = begin next;
C:             call start_subtree(root);
D:             repeat deactivate next(null);
            end next;
         end only_known_entry;
      end infix; ...
```

Figure 4.5.4 A recursive coroutine as generator function.

The effective entry value of "next" is initialized to refer to the prologue of the procedure body of co_function "next". Thus, "next" is a creation entry for a newly created instance of "infix". Traversal of the tree is initiated due to the statement "C" which invokes "start_subtree". The effective entry value of "next" is set to refer to statement "B" whenever deactivation occurs due to elaborating statement "A", thus avoiding the reelaboration of the prologue of the co_function "next". Thus, "next" becomes an activation entry when control

passes through the statement "A" for the first time. Once the traversal has been completed, "start_subtree" returns to the original caller, i.e. statement "C". Elaboration proceeds then to statement "D" which will deliver a null value for this and all subsequent activations through "next". Thus, every instance of the "infix" traverses the underlying tree once and only once.

5. Information Transmission.

In this chapter we will study the problems relative to information interchange between modules. In particular we will examine how independently compiled modules may interchange information.

Information may be interchanged in several ways, e.g. by means of global areas, actual/formal parameter lists as well as by means of message transmission. We will show that, from the point of view of actually making information available, we can study these three different forms of information interchange in a unified manner.

The basic concept which we will use for interchanging information is the parameter list. We will define parameter list association in such a way that it becomes explicit which parameters are to be associated. As we shall see, this allows us to perform piecemeal association of parameter lists. It also allows us to dissociate the textual order of parameters from the order in which parameters occur in parameter lists. That is, we will depart from the positional association rule of parameters which is common to most of the present day languages. Finally, due to our parameter list association rule, it becomes possible to define one single parameter list which contains formal, global as well as message (receiver) parameters.

Besides parameter lists, we will also study module typed variables and the activation forms of modules. Thus, we will study control flow and data flow driven modules. Furthermore, we will relate interrupt handlers with data flow driven modules. A deeper treatement of interrupt handlers will be left for chapter 6.

Module typed variables pose several problems of their own. We are particularly interested here in defining association of parameter lists in the presence of module typed variables. As we shall see, such module typed variables imply that association will act upon parameter list typed variables. Furthermore, module typed variables imply the need for template parameter lists in addition to actual and formal parameter lists. Such a template parameter list describes the parameter list related to the uses (calls) of module typed variables.

Module typed variables will be treated in the same way as any other variable. It is thus valid to define arrays of modules or functions which return module typed values.

This chapter is subdivided into 4 sections. In section 5.1 we will study parameters as individual entities. In particular we will study the operations of association and transmission in this section. Furthermore, we will define what we understand as control flow and data flow driven modules in this section.

In section 5.2 we will study parameter lists. We will define in this section a parameter list association mechanism which we claim allows more freedom and provides more accuracy with regard to association. We will show also in this section, that the same association mechanism serves for generating values of a composite type, e.g. structure typed values.

In section 5.3 we will study module typed variables and parameter list typed variables. We will show how association can be performed using the same rule as defined in section 5.2. We will study also how gradual association of parameter lists could be achieved.

Finally, in section 5.4 we will study several independent problems related to information interchange. In particular we will study IPL-V's information interchange mechanism; the Jensen device; and composition of modules into packages, i.e. groups.

5.1 Association and Transmission.

In this section we will study the basic operations which are required to accomplish information interchange between modules. We will study these operations in a uniform manner. That is, we will not differentiate between information interchange achieved by means of local/global, actual/formal parameters and that obtained by means of message transmission. Finally, we will study these operations from the point of view of independently compiled modules.

Information interchange implies the existence of storage, i.e. a sending and at least one receiving data space. Notice that the sending "data space" could be a function which produces the desired information whenever this function is activated, e.g. "call by name"[Gri1]. As has been shown in chapter 2, data spaces are characterized by names. Now, in order for the programmer to obtain such a name, a locator function of some textual name, say of a parameter, must be elaborated. Parameters may be of any type, e.g. primitive type, linked list type, procedure type etc. In many cases parameters will stand for access typed variables, but even then the set of target types is not restricted. Notice that from this point of view we could have global parameters called "by name".

Observe that parameters could have been generated by the language processor and, consequently, are hidden from the user. Consider for instance "call by result" in ALGOLW. Here a parameter is generated by the language processor. This parameter is the one which will be used for information interchange with the "calling" module. Within the "called" module a local data space is bound to the user defined parameter. Finally, at the end of elaboration of the "called" module, an epilogue is elaborated which transmits the contents of this local data space to the language processor generated parameter and, consequently, back to the calling procedure.

For each effective information transmission there must be a sending and a receiving parameter. In many cases the roles of these parameters may be reversed from transmission to transmission. For instance, some transmission lines allow the transmission of messages in either direction. Similarly a "call by reference" may cause side effects, i.e. "transmissions" from the formal to the actual parameter. It follows then, that the property of a parameter being sending or receiving can be established in the general case only when transmission actually occurs.

We will call <u>parameter association</u> the operation of relating two or more parameters for the purpose of information transmission. Notice that parameter association does not imply actual transmission. It does imply, however, that the data spaces required for the transmission exist, that they are bound to their respective parameters and that these parameters are known to the module which eventually will perform the transmission. Thus, once two or more parameters have been associated, zero or more effective transmissions may occur.

Parameter association could be of either of following two forms:

i- explicit- the parameters involved in the information interchange are bound to different data spaces. An information transmission implies thus an actual information transfer.

ii- implicit- the parameters involved in the information interchange are bound to one and the same data space. There is then no actual information transfer between parameters. Notice though, that any access, be it a read or a write access, corresponds to a "transmission" in this case.

Transmission simply copies the contents of a data space to another without performing any conversion. It follows then, that the parameters associated by means of an explicit association must satisfy lemma 3.5.3 [sequentially multityped binding conditions] in order to maintain the sequentially multityped environment. In the case of implicit association, the associated parameters must also satisfy lemma 3.5.3, since such parameters are bound to the same data space. Formalizing we have:

Lemma 5.1.1 Parameter association maintains the sequentially multityped environment, iff the associated parameters satisfy lemma 3.5.3, where read access means transmission from a sending parameter, and write access means transmission to a receiving parameter.

In ALGOL60 formal parameters do not have to be explicitly typed in the case of "call by name"[viz. 5.4.5 in Nau2]. This does not contradict lemma 5.1.1 since either

the receiving module must be recompiled (substitution rule taken to an extreme) or what is passed is a function, i.e. thunk[Ing1], which precedes or succeeds the transmission with an appropriate conversion. Observe that this is a case where transmission is performed by providing an interfacing procedure. That is, each transmission consists of two steps and necessarily passes through the interfacing procedure.

Let A and B be two textual names and let LF_A and LF_B be the respective locator functions. Suppose that the name n was first bound to LF_A. Let P be the set of input values, say the binding information, which, when given to LF_B, causes LF_B to produce the name n. For example, P could be the name typed value n, and LF_B an indirect access through the data space containing P. It follows from lemma 5.1.1 that, in order to assure access correctness, the input values P must be abstracted from LF_A and the input values used by LF_A. Consequently the input values P must be transmitted from the module containing LF_A to the module containing LF_B. Furthermore, the module which abstracts P must perfectly understand LF_A, LF_B and the input values to LF_A, since, otherwise, P could not possibly be abstracted. This knowledge could have been established by convention, say by the binding convention. If not known by convention, it follows that this knowledge must have been gained by means of some previous transmission. We have then:

Lemma 5.1.2 Implicit association is possible, iff the locator functions of the parameters being associated are known by binding convention, or if they have been made known by means of a previous transmission.

Implicit association can be viewed then as a binding operation which acts over the boundaries of two or more

modules. Furthermore, this binding is achieved by sending binding information from one module to another. Formalizing we have:

Lemma 5.1.3 Previous to any implicit association, an explicit association must have occurred, such that the binding information required by this implicit association can be obtained by means of one or more transmission steps.

In order to perform an information transfer, the names of the sending and the receiving data spaces must be known and bound to the corresponding parameters. This knowledge could be established by means of some convention, e.g. a standard "calling sequence". This convention will be called existence convention. It differs from the binding convention in that there is no need to bind any parameters in this case.

Names of data spaces are established within the module possessing these data spaces. Thus, except for the case of an existence convention, these names, i.e. the binding information corresponding to these names, must have been sent to the module performing the association. Explicit association can be viewed then as an information gathering operation. From this discussion we may conclude:

Lemma 5.1.4 Explicit association is possible iff the binding information of the parameters involved in the association is known by existence convention, or it is made known by means of a previous transmission.

Let C and D be two parameters. Suppose that C and D are implicitly associated, i.e. C and D are bound to the same data space. It follows from lemma 5.1.3 that there

must be two parameters N and R which interchange the binding information necessary to associate C and D. Suppose now that C and D are explicitly associated. Suppose, furthermore, that there is no existence convention which allows C and D to be associated immediately. It follows from lemma 5.1.4 that also here there must be two auxiliary parameters R and N which interchange the binding information in order to accomplish the association of C and D. Suppose, finally, that C and D are explicitly associated and that there is an existence convention which establishes this association. Now, the existence convention defines in fact the value of the binding information which permits the association of C and D. It follows then, that the auxiliary parameters N and R exist but can be omitted since the information they interchange is known statically. We may conclude then:

Fact 5.1.5 Information interchange is accomplished by means of following four parameters:

i- a root parameter R which contains the binding information to be sent;

ii- a non-root parameter N which is associated with a root parameter and receives the binding information;

iii- a carrier parameter C related to a root parameter R and which is used to perform the actual information interchange;

iv- a dummy parameter D related to a non-root parameter N and which is associated with a carrier parameter C.

In figure 5.1.1 we show an example where the four aforementioned parameters are typified. Observe that the occurrences of the dummy parameter "P" in the statement

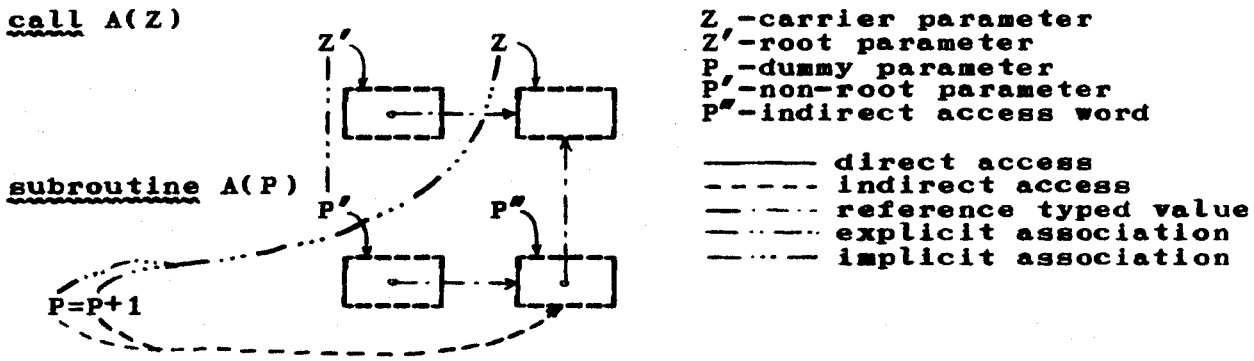


Figure 5.1.1 FORTRAN IV parameter association using indirect access.

"P=P+1" are implicitly associated to the carrier parameter "Z", since the locator function of "P" is bound to the data space characterized by "Z". Notice that the locator function of "P" is an indirect access via "P"", thus different associations can be accomplished by simply changing the value of "P"". The non-root parameter "P'" exists solely to enable the copy of the value contained by the root parameter "Z'" to the indirect reference word "P"". Thus the existence of "P'" may well be hidden in the code generated by the FORTRAN IV compiler.

The explicit association operation does not necessarily occur within the sending module. Consider for example "call by reference" implemented by means of "address plugging". In this case all instructions which refer to the formal parameter must be initialized to refer to the actual parameter's data space previous to the elaboration of the module. Thus there is effectively an information transmission. Furthermore, the corresponding parameter association is explicit. Now, the instructions to be initialized are known only to the module containing them. It follows then, that the association operation and, consequently, the

transmission operation must be performed within this particular receiving module.

It should be clear now that association is possibly a compound operation involving several information transmissions in order to be accomplished. Furthermore, association depends on the binding and existence conventions. Thus, a bad choice for these conventions may entail a decrease of efficiency of the running program. Observe that carrier and dummy parameter association implies that the corresponding root and non-root parameters have previously been associated and that some information transfer between these parameters has been accomplished.

Notice that there could be several call conventions, e.g. "by reference" and "by name" conventions. Now, if the parameter type also indicates the convention, it follows that, by means of type checking, we could assure association to be convention-wise correct. Notice that this is nothing more than to establish a higher level convention by means of which specific lower level conventions, such as "by reference" or "by name", could be abstracted. Thus, if the language provides for constructs by means of which we could specify lower level conventions, the problem of choice could become adaptive to the class of programs to be produced.

We have frequently refered to actual/formal and local/global parameter associations. These terms only define what we will call association classes. Each of these classes characterizes whether association is implicit or explicit and whether it occurs statically or dynamically. Thus, local/global and actual/formal parameter associations imply implicit association, whereas message transmission, i.e.

sender/receiver association, implies explicit association. Now local/global association differs from actual/formal association in that association is performed statically in the former case and dynamically in the latter case. Notice that this is the case in block structured languages, since there the association operation is in fact a locator function copying operation. In the case of sender/receiver association, there is no need for a partitioning relative to association time, since static association implies that the binding information is transmitted statically, thus it could be known by existence convention.

A parameter is said to be *filled* whenever it is bound and the data space it characterizes contains valid, i.e. defined, information. Observe that a filled parameter does not necessarily stand for up to date information.

Information transmission requires time in order to be accomplished. During this time interval the data space which will eventually carry the information transmitted, is not completely known in some cases, although the textual name referring to this partially known data space may be bound and the data space may contain defined information, e.g. when copying a list. It follows then, that fill checking may succeed even when the information has not been completely received. Since we are also unable, in the general case, to determine when a transmission has been completed, it follows that there must be a *data presence* switch associated with each receiving parameter. This switch will be *off* as long as the information has not been transmitted completely. It is set to *on* after the information has been transmitted completely. Observe that setting the data presence switch to *on* must be performed by the transmission operation itself, since only it knows when the

transmission has been completed. This shows also that there should be a language constructs which allows declaring a sub-module to be a transmission module.

Data presence switches may be set implicitly. This is the case whenever it can be assured by means of some convention, that any information transmission has necessarily completed when the receiving module is activated. Copying a "by value" parameter is an example of this case. In the case that no data presence switch is explicitly present and there is no implicit setting of data presence switches, parameters must be fill checked, since otherwise uncontrollable accesses might occur. From the preceding discussion it follows though, that fill checking does not assure that the information contained in the parameter's data space is up to date. Not only this, the information contained by this data space may be misleading and eventually cause severe errors, e.g. when containing incorrect data space identifications. This shows then a need for a data presence convention by means of which such errors could not possibly occur.

In order to reduce overhead, data presence checking or fill checking should be performed as few times as possible, preferably statically, or, even better, implicitly. The most adequate points to perform such tests are algorithm dependent and, consequently, must be established by the programmer. This suggests then the existence of a construct which allows to make the data presence checking explicit for a set of statements, e.g. a compound statement. Thus, if this compound statement can be activated only when the data presence check succeeds, any access to the checked textual name is necessarily correct, as long as the data presence status cannot be changed by some other module

executing in a parallel or quasi-parallel form. We will come back to this later in this section.

A module is said to be data flow driven[Rod1,Slu1, Kos1,Kos2,Dav1], if its activation is triggered whenever the data presence switches of all receiving (dummy) parameters are on. Conversely, a module is said to be control flow driven if activation and data presence checking are independent operations for all receiving parameters used by this module. Notice that there could be modules which are partly control flow driven and partly data flow driven. For example, the PL/1 ON ERROR interrupt handler is activated whenever an error condition is flagged during elaboration. Thus the presence of an error information is sufficient to trigger the activation of the corresponding interrupt handler, regardless of whether it has been bound to a user defined on-unit. Notice that defining an on-unit corresponds to an implicit parameter association, i.e. the interrupt handler is bound to the user defined on-unit. Recall that parameters may stand for values of any type, in particular they may stand for procedures.

It should be clear that, within control flow driven modules, activation may occur when some of the dummy parameters have not been associated or their data presence switch is off. This is not necessarily an error condition, since it may be known beforehand that, for a given set of input values, the module will never trace a code section requiring the missing information. It follows then:

Fact 5.1.6 Forcing all information to be present when activating a module restricts the power of expression of the language.

There are two ways of maintaining access correctness in the case of missing information:

i- precede all effective uses of a dummy parameter by a presence checking operation which determines whether the information is present or absent, i.e. whether the access can be performed.

ii- define a default parameter value to which a dummy parameter P is initialized whenever P has not been associated or the corresponding data presence switch is off.

There are some languages which allow the definition of default parameter values. For example, keyword macros in the IBM/360 Assembler F[IBM4] allow the definition of such values. As mentioned before, PL/1 on-units must not necessarily be user defined, i.e. system defined on-units are another example of default parameter values. Observe that presence checking implies binding checking in the implicit association case.

Notice that default parameter values could be variable and thus be made dependent on previous activations. This seems to be a better approach to ALGOL60 own variables since, in this case, they might contain defined values even for the very first activation, thus eliminating the requirement for an external driving data space which contents determine whether the own variable contains an undefined value or not. In section 5.2 we will show that there is no necessity for a bijection between the actual and formal parameter lists, thus the existence of own variables could even be hidden from the the user of the module defining such own variables.

There are two cases of data flow driven modules to consider:

i- asynchronous- in this case activation of the module occurs whenever all data presence switches are on. Notice that this case implies parallel or quasi-parallel (multi-programmed) elaboration of modules.

ii- synchronous- in this case also all data presence switches must be on, however, the program must also be in a state, say activation state, which allows the activation of the module.

Notice that we are considering synchronization relative to the sequence of states a program passes through, and not relative to a program independent clock pulse.

Let SD be a synchronous data flow driven module. There is then a set of activation states $S=\{s_1,s_2,\ldots,s_n\}$ such that SD could be activated at some state s_i in S. Furthermore, SD can be activated only if all data presence switches are on. Thus, either the data presence check is conducted at state s_i, or s_i is enabled to trigger the activation of SD after all the data for SD has been found to be present. Now, enabling s_i to trigger the activation of SD, is nothing more than to increase the data dependency of the module SD to include a "currently at state s_i" dependency. Hence the data presence check is completed only at state s_i. But then SD can be simulated by means of a control flow driven module which activation at state s_i is preceded by an adequate data presence check. We have then:

Fact 5.1.7 Synchronous data flow driven modules can be implemented by means of control flow driven modules.

In order to decrease the amount of interface information and, consequently, decrease the possibility of incorrect module interactions, we would like the data presence checking to occur only within the simulating control flow driven module. That is, at state s_i the simulating module M is activated regardless of the data presence switch settings. However, the activation will succeed only if all switches are on.

Defn. 5.1.8 A module M is said to be *data presence transparent*, if the data presence checkings performed by M are unknown externally to M.

If M is to be a data presence transparent module, it follows that M must receive at least two effective return values which are:

i- success return- this is the elaboration resumption point in the case that the data presence check succeeds. Thus a return via this effective return implies that the data flow driven module has been elaborated successfully.

ii- failure return- this is the elaboration resumption point in the case the data presence test fails. It is usually different from the success return, since successful and unsuccessful elaboration of a module normally determine different subsequent actions.

Notice that these effective returns do not necessarily have to be return labels in the usual sense. That is, such an effective return could be the activation point of some other module, e.g. coroutine. We have mentioned in section 4.5 that multiple effective returns could be implemented by means of a return point structure. Furthermore, the deci-

sion of which effective return is to be used relies with the returning module. Finally, the necessity of transmitting a return point structure can be tested when combining two modules by means of a simple interface information test. Thus this problem can be effectively converted into a type checking problem.

Instead of returning via different effective return points, the module could deliver a value of some type, e.g. a boolean. This value would determine then whether the elaboration was successful or not. Observe though, that this represents a duplication of testing effort, since success or failure is tested within M itself in order to determine the value to be returned, and, externally to M, the returned value is examined in order to determine whether M succeeded or failed. This shows then, that providing the ability of defining return point structures possibly reduces the error proneness of the language and increases both the structuring and the efficiency of the programs.

Let CM be a data presence transparent control flow driven module instance which simulates a synchronous data flow driven module instance DM. Since CM is control flow driven, CM is activated from a module instance AM, where AM≠CM. Since CM is data presence transparent, CM possesses at least two sub-modules T and E, where T performs the data presence checking, and E actually simulates the data flow driven module DM. Since CM is used in place of DM, any instance of T or E must be bound to CM. Thus, in order to activate a submodule T, first the module instance CM within which the instance of T is to exist must have been created. It should be clear that this could be implemented using the same constructs as those needed to implement the generator functions introduced in section 4.5. That is, the module

instance CM is obtained by a (possibly dynamic) module instance declaration, whereas creations of instances of T and E are qualified with the module instance CM within which they exist. We have then:

Fact 5.1.9 The constructs necessary to implement generator functions are sufficient to implement data presence transparent control flow driven modules which simulate data flow driven modules.

It should be clear that, in many cases the set of program states which may activate a data flow driven module is quite large. Simulating data flow driven modules by means of control flow driven modules may thus entail a high degree of inefficiency due to repeated unsuccessful elaboration of the data presence checking sub-module. Consequently it would be worthwile to develop language constructs from which those program states which imply successful data presence checking could easily be abstracted. Such constructs have been developed for several data flow driven programming languages[Kos1,Dav1,Kos2]. We will not enter into further details with respect to these languages though.

Consider now the case of external interrupts. Such an interrupt could be regarded as a piece of information. It has furthermore the property of occurring at unpredictable times. If interrupts could be disabled, or if they have a priority of service associated with them, there is clearly a state dependency with respect to servicing these interrupts. Even in these cases the interrupt handlers are usually considered as being asynchronous data flow driven modules, since the set of activation states is almost as large as the complete set of program states. It follows

then, that simulating interrupt handlers by means of control flow driven modules may become quite costly.

Data flow driven modules could be simulated by means of interrupt handlers, where the interrupt condition is the presence of all information necessary to activate the data flow driven module. It follows from this that interrupt handlers may depend on several parameters, rather than just on one as PL/1's on-units. In order to reduce overhead, the interrupt condition should be tested as few times as possible. Thus, it should be tested only when any of the interrupt handler parameters is changed. This implies then, that we must emulate all store operations which store into data spaces which are also interrupt handler parameters. We will study interrupt handlers in greater detail in chapter 6.

Once two parameters have been associated, the sending and receiving data spaces are known to the module which will perform the information transmission. This does not imply though, that the transmission is permitted. There are two cases which forbid information to be transmitted:

i- the information contained in the sending data space is not yet up to date. This is the case when a data space is shared by several concurrently elaborating modules, e.g. critical regions[Hoa8,Han3]. Recall that any access to an implicitly associated parameter corresponds to a data "transmission".

ii- the information contained in the receiving data space is still meaningfull, i.e. may not be overwritten. This is the case when transmitting information by means of a buffer[Han3]. As long as the information in the buffer has not been consumed, no buffer

filling should occur, since this could cause the loss of valuable information.

It follows from this that following switches are required:

i- sending status- this switch is set to ready iff the information contained within the sending data space is up to date.

ii- receiving status- this switch is set to consumed iff the information contained within the receiving data space may be overwritten without causing loss of valuable information.

Furthermore, any transmission must be preceded by a ready test which verifies whether the sending status is ready and all receiving stati are consumed. Clearly this check could be performed once for several actual transmissions, as it is the case in critical regions. It could also be performed implicitly, i.e. by convention, as it is the case in single-programmed languages such as ALGOL60, SNOBOL4 etc.

In fact there are at least two tests, one for the sending status and the others for the receiving stati. Since testing and transmission takes time, it follows that, during this time interval, other modules should be prevented from changing the sending and the receiving stati.

Defn. 5.1.10 A transmission module M is said to be externally indivisible if, whenever M is being elaborated, no other module Q may change the sending status of the sending data space and/or access the receiving data spaces affected by M.

Observe that critical regions are examples of such externally indivisible modules. It should be clear that the sending status could be implemented by means of semaphores[Dij2,Dij3,Hoa8,Han3]. Semaphores and the data spaces they control are disjoint though. This leads easily to incorrect elaborations due to inadequate settings and/or resettings of the sending status, i.e. semaphore. Observe though, that by means of the shared variables proposed by Hansen[Han3], this problem can be reduced to a type checking problem. This follows from the fact that a shared variable can only be accessed from within a critical region. Furthermore, the critical region depends on the sending status associated with this particular shared variable. That is, access can be effectively prevented whenever the sending status is not_ready.

In order to allow transmission, the receiving status must be consumed. Furthermore, it should be set to not_consumed once the transmission has been completed. It follows then, that the receiving status is equivalent to the data presence switch mentioned earlier. It follows also that the data presence switch should be set off once the information has been consumed. Clearly this must be performed by the consuming module, e.g. a user's program.

Suppose now that the receiving status is implemented as a semaphore. From the definition of ready check it follows that transmission may occur only when all modules which could access the receiving data space are prevented from doing so. Thus in a sense, we are dealing here with an "inverse" semaphore. That is, access is granted to the transmission module only when access in general is denied. Thus, it is not the transmission module which denies other modules to access, but the fact that other modules are

prevented from accessing is what allows activation of the transmission module. This stresses the fact that transmission modules should be identified as such.

The preceding discussion shows the need for a consumption section construct, which allows activation only when the required information is present. Furthermore, after completing elaboration it sets the receiving status to consumed. In the case of several consuming modules to a given receiving data space, the receiving status must be individualized per module. If it were not done so, a same consuming module could be activated several times for the same piece of information.

Data flow driven modules and parallel elaboration of modules are closely related[Kos1,Kos2,Dav1]. With respect to parallel elaboration there are further difficulties to overcome, such as deadlocks[Hol1,Hol2,Hol3,Hab1] and infinite delays[Dij7,Hoa8]. We will not deal with these difficulties though, since this is beyond the scope of this dissertation.

5.2 Parameter Lists.

In the previous section we have discussed the case of simple parameter association. In many cases though, we want to associate several parameters by means of only one textual association, e.g. when associating actual/formal parameter lists. In this section we will study how this multiple association could be achieved so that:

a- the possibilities of association errors are reduced;

b- there is no necessary one to one relation between the parameter lists;

c- the concept of data transmission can be extended to the generation of values of composite types.

A parameter list is a set of parameters. This set is not necessarily ordered. Usually language processors store sets as if they were ordered though. There is then a parameter list implementation which is an ordered set and every element of this ordered set stands in an one to one correspondence with an element of the corresponding parameter list.

Parameter lists are types and they are kept within some data space. Furthermore, there is a textual name which identifies this data space. Usually this textual name is the name of the module, i.e. procedure, to which the parameters are passed. We will allow, however, that parameter lists be identified by textual names which do not have to identify modules too, e.g. entry names of multiple entry modules; the name of a named COMMON list in FORTRAN IV etc.

Carrier parameters may be represented textually as expressions or sequences of statements. Thus, before transmitting information, these expressions must have been elaborated. We will call this elaboration carrier parameter elaboration. Observe that the result of the carrier parameter elaboration may be the binding information which is to be transmitted to the non-root parameter. This occurs, for instance, when the result of the carrier parameter elaboration is an access typed value, e.g. "call by reference".

It follows from fact 5.1.5 that for each carrier parameter there is at least one root parameter. Furthermore, for each dummy parameter there is at least one non-root parameter. In order to associate a carrier and a dummy parameter list, the corresponding root and non-root parameter lists must be associated first. Not only this, the association of the root and non-root parameter lists determines the association of the carrier and the dummy parameter lists. It suffices then to study in detail the root and non-root parameter list association.

Parameter lists are represented textually as carrier or dummy parameter lists. Since the parameters in the root (non-root) parameter lists do not necessarily stand in an one to one correspondence to parameters in the carrier (dummy) parameter lists, it follows that we must be able to specify the root (non-root) parameter lists and their relation to the corresponding carrier (dummy) parameter lists in some cases.

When associating parameter lists, elements of these lists must be associated. Thus there must be an <u>association rule</u> which associates one element of the root parameter list with zero or more elements in the non-root parameter list. It follows from this definition, that there could be root parameters as well as non-root parameters which will remain unassociated. This occurs for example in SNOBOL4.

Observe that, in order to start association, the module performing this association operation, e.g. procedure prologue, must know the parameter lists to be associated. This implies then an information interchange by means of which the parameter lists to be associated are made known to this particular module. Notice that this information

interchange takes a parameter list as if it were a single value. Furthermore, in order to appropriately associate elements, the organization, i.e. type descriptor, of each of these parameter list types must be known. Usually the organization of a parameter list is known by convention, e.g. standard calling sequence.

The association rule is frequently based on the notion of ordered sets. That is, the parameter lists are assumed to be ordered sets and elements in the same ordinal position are to be associated. Observe that the association could occur statically in this case and, consequently, the run time cost of the program is reduced. Furthermore, carrier and root parameter lists as well as dummy and non-root parameter lists stand in an one to one relationship in this case, consequently, the root and non-root parameter lists may be implied. This positional association rule has several drawbacks:

a- it is relatively error prone, mainly when the parameter lists are large (>5), since a single omission error may result in multiple association errors, and, also, permutations of parameters are not mechanically noticeable in most cases, e.g. when the permutation occurs with respect to equally typed parameters;

b- the order of the parameter lists is usually abstracted from the text. In procedural languages this imposes a carrier parameter elaboration order which is not necessarily the desired one.

This suggests then an alternate association rule, where the parameters to be associated are explicitly specified. Observe that this explicit specification is possible

since the parameter lists must be known in order to achieve a meaningful information interchange between modules.

In section 3.4 we have briefly introduced the concept of parameter names. Since it is the programmer's ultimate responsibility to determine which parameters are to be associated, it follows immediately that parameter names are some sort of textual names which identify individual parameters. Furthermore, their textual scope is restricted to parameter lists, thus they may be declared by means of context, e.g. "A::". Finally, each parameter name related to a carrier (dummy) parameter defines a root (non-root) parameter bearing this particular parameter name and which sends (receives) the binding information relative to this carrier (dummy) parameter.

Notice that such parameter names occur in some programming languages. For example, they occur in keyword macros in the IBM/360 Assembler language[IBM4]. It is the author's understanding that the ICL ATLAS FOTRAN V "PUBLIC" spaces also provides such names and that association takes these names into consideration[Sch2].

Defn. 5.2.1 The named association rule is an association rule which associates those and only those parameters bearing the same parameter name.

Since transmission and binding can occur with respect to one carrier parameter only, we have:

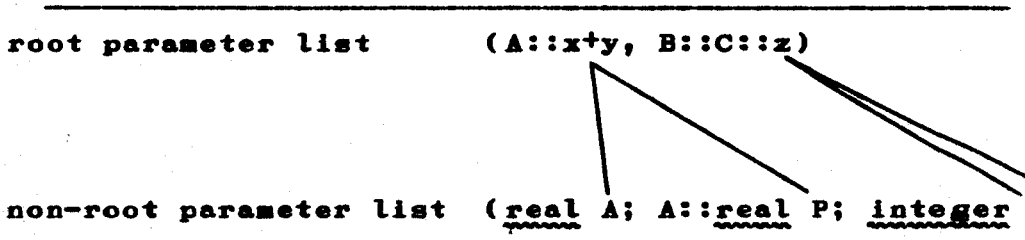
Lemma 5.2.2 All parameter names within a carrier parameter list must be mutually unequal.

Notice that lemma 5.2.2 implies that also all parameter names occurring within root parameter lists are mutually unequal. This inequality requirement does not

necessarily apply to dummy parameter lists, since it is valid to associate several dummy parameters with one carrier parameter. Dummy parameters are themselves some user defined textual name. Thus, in order to diminish the writing effort, it follows that this textual name could also function as the non-root parameter parameter name. Now, in order to allow several non-root parameters to be associated with only one root parameter, we must be able to define alias names for non-root parameters. Observe that we are not concerned here with the problems induced by the existence of multiple access paths to a same data space.

It follows from the named association rule that the root and non-root parameters to be associated must bear the same parameter name. Thus, the parameter name of the root parameter must be equal to the name of the corresponding non-root parameter, or one of its aliases. Now, the root parameter name is abstracted from the carrier parameter list. To constrain the textual names of the carrier parameters to be equal to the parameter names of the non-root parameters is too strong a restriction, for it restricts the freedom of naming in such a way that it may become impossible to combine modules without completely rewriting them. It follows then, that parameter names should be provided explicitly within the carrier parameter list.

In figure 5.2.1 we show pictorially how our named association rule works. There are three root parameters, "A", "B" and "C". The root parameter "A" is related to the carrier parameter expression "x+y". The root parameters "B" and "C" are both related to the carrier parameter "z". There are also three non-root parameters, "A", "B" and "C". The non-root parameter "A" is related to both the dummy



The arrows show the parameters which are to be associated.

Figure 5.2.1 Pictorial description of the named association rule.

parameters "A" and "P". The latter is due to the alias definition "A::real P". Finally, the non-root parameters "B" and "C" are related respectively to the dummy parameters "B" and "C". Observe that the non-root parameters are defined implicitly, whereas the root parameters are defined explicitly.

In this section we are considering only direct associations. That is, we are not considering the case of parameter list typed "variables". Parameter list variables will be studied in section 5.3. Any association which associates more than one root parameter with one non-root parameter is clearly in error. Since both parameter lists must be known and the root parameter list can adapt to the non-root parameter list, it follows that there is no loss of generality in the direct association case when enforcing that at most one parameter name exists per non-root parameter. Now if this is the case and also lemma 5.2.2 holds, it follows immediately that there could not possibly be any of the aforementioned association errors. Using the syntax of figure 5.2.1 we will assume then, that the leftmost alias name is the parameter name of the non-root parameter. For example, in figure 5.2.1 the dummy parameter "P" is known externally, i.e. as non-root parameter, only by means of

the alias parameter name "A". Thus, even if a root parameter with parameter name "P" would have been defined, there would be no association error, since there is effectively no parameter name "P" within this non-root parameter list.

To implement the named association rule offers no major difficulties. It might be costly though, since some processing may be required in order to determine which of the non-root parameters are to be associated with a given root parameter. Now, if we can assure that the association rule is necessarily a bijection between the parameter lists being associated, we could devise an ordering rule and, thus, effectively transform the named association rule into a positional association rule. For example, the language processor could order the parameter list implementations according to the parameter names. If the association is bijective, it follows that such an order exists, that it is unique and that both lists are equally ordered. Observe that in many programming languages the association is in fact bijective, e.g. ALGOL60 and ALGOL68. Notice also that we could partition parameter lists into several sections, one of which allows a bijective association.

It follows from above discussion that even in the case of the named association rule, the association could be performed statically whenever the association is bijective and is known statically. Now, in case the association is neither bijective nor statically known, processing is required anyhow when performing the association, since we must determine all those parameters which are not to be associated or which are to be multiple associated. We have then:

Fact 5.2.3 The named association rule does not necessarily cause a cost increase over the positional association rules

Carrier parameter elaboration may cause side effects. For example, there are functions which change their internal state whenever they are activated, e.g. generator functions. Now if such a function is activated when elaborating a carrier parameter, a side effect occurs. It follows then, that the order in which carrier parameters are to be elaborated must be specified in some cases. Furthermore, this order is not necessarily the order in which the corresponding parameters occur within the carrier parameter list implementation. Of course, the order of elaboration could be artificially established by preceding the association operation by several statements which elaborate the carrier parameters in the desired order and assign the result to some temporary data space. Thus when associating, the names of these temporary data spaces could act as the carrier parameters, thus not requiring any elaboration at all. However, we consider this solution as being inelegant due to the artificial introduction of temporaries. Not only this, it is usually no longer clear from the text of the carrier parameter list what values are effectively being passed. We may claim then, that this solution represents a decrease of program structuring. Now, by means of the parameter names, the language processor could place the elaborated carrier parameter into the appropriate order within the carrier parameter list implementation. It follows then, that the textual order in which modules (statements) which elaborate the carrier parameters occur in the program is independent from the order in which

parameters occur within the carrier parameter list implementation when parameter names are used.

By means of parameter names it is possible to group parameters and to produce parameter hierarchies. This has several advantages:

a- logically interdependent parameters can be grouped and this can be made explicit within the program's text;

b- it is possible to build parameter lists by combining several parameter lists together, e.g. in the case of procedure typed variables as we shall see in section 5.3;

c- structured carrier parameter lists allow the generation of values of composite types.

```
DCL 1 A,
      2 B,
      2 C,
        3 D,
      2 E;

A := (B::x, C::(D::y), E::z);
```

Figure 5.2.2 Example of an assignment to a PL/1 structured variable.

In figure 5.2.2 we show how a PL/1 structured variable value could be generated by means of a hierarchical carrier parameter list. Observe that this form puts into evidence which values are to be transmitted to which dummy parameter, i.e. element variable.

Observe that it is syntactically quite easy to determine the type of the composite value to generate if some

rules of operand type dominance are provided. In the case of figure 5.2.2 the type of the variable at the left hand of the assignment operator defines the type of the value to be generated at the right hand side.

We may desire that the order of elaboration of the carrier parameters be different from the implementation order of the parameter list hierarchy. It follows then, that we must be able to provide the parameter name by means of a qualification list.

```
DCL 1 A,
      2 B,
        3 C,
        3 D,
      2 F,
        3 G,
        3 H;

A := (B.C::x, F.G::y, B.D::z, F.H::w);
```

Figure 5.2.3 Example of an assignment to a PL/1 structured variable using qualified parameter names.

In figure 5.2.3 we show a case where the order of elaboration of the carrier parameters, first "x" then "y" then "z" then "w", cannot be mapped onto any hierarchy preserving permutation of the underlying structured variable. Thus this is a case where we require the parameter names in the carrier parameter list to be explicitly qualified.

Concluding this section we want to point out that the dummy parameter lists contain formal, global as well as receiver parameters. Notice also that returning of function values can be considered as an actual/formal association, e.g. "call by return". Finally, the non-root parameter list description is part of the interface information of the module containing this list. This description defines the

non-root parameter names and also the types of the dummy parameters related to each individual non-root parameter. Observe that there is in fact a description hierarchy, since the parameter lists itself is a value of a given type and, hence, must be described within the interface information.

5.3 Parameter List Typed Variables.

In the preceding section we have studied the association of constant parameter lists. In this section we will study the association of parameter lists contained within parameter list typed variables. We will show that module typed variables imply parameter list typed variables. We will introduce also the concept of template parameter lists which are required in order to allow the transmission of module typed values across module boundaries.

It should be noted here, that root and carrier parameter lists are treated as a single entity. Similarly non-root and dummy parameter lists are single entities. In section 5.2 we have already mentioned that it suffices to study root and non-root parameter lists association, since this association implies the association of the corresponding carrier and dummy parameter lists. Furthermore, association of carrier and dummy parameter lists implies that the associated carrier and dummy parameters are bound, possibly to the same data space. In order to avoid ambiguities, we will specify which parameters are being associated when using the term "association".

Consider the case of associating a constant root parameter list with a variable non-root parameter list.

This case occurs with respect to procedure (module) typed variables (parameters). This follows from the fact that each module typed value which a given module typed variable could contain defines a different dummy parameter list and, consequently, a different non-root parameter list. Thus the non-root parameter list related to a module typed variable changes as the contents of this variable change.

In section 3.4 we have shown that type descriptors must be transmitted between modules. As a consequence also the operators defined by these type descriptors must be transmitted. Now, these operators are in fact procedure (module) typed variables (parameters). We have then:

Fact 5.3.1 Preventing the existence of module typed variables restricts the power of expression of the languages

We will consider module typed values in the same way as any other value. That is, we allow the existence of module typed assignment [e.g. ALGOL68 proc variables]; functions which return module typed values [e.g. LISP1.5 and GEDANKEN]; module typed expressions; arrays or structures containing module typed values; references which target type is "module" [ref proc type] etc.

We will proceed now to define some symbols which, hopefully, will aid in reducing ambiguities within the subsequent discussion. Let P be a module which contains a module typed variable V. This variable V may hold any of the module typed values in the (possibly infinite) module set $Q=\{Q_1,Q_2,\ldots,Q_m\}$. Each of these module typed values Q_i defines a non-root parameter list N_i. Let S be a statement in P which associates a root parameter list R with the non-root parameter list N_i of the module Q_i contained in V. Observe that S does not imply that elaboration of Q_i will

be started whenever S is being elaborated. Furthermore, the association and filling of parameters in the non-root parameter list N_i could occur piecemeal at several different instants. Recall that textually N_i is represented as a dummy parameter list D_i and that R is represented textually as a carrier parameter list C.

From the description of N_i above it may appear that we are allowing only one dummy parameter list per module. Although this is the usual case, there could in fact be more than one dummy parameter list per module. In this case some dummy parameter list identification must be present in statement S in order to assure correct association of parameters, e.g. entry point name; named COMMON etc.

```
begin module_P;
   declare procedure V;
         ...
   V:=Q_i;  /* assign a module typed value to V */
         ...
S: associate V:C;
         ...
end module_P;
```

Figure 5.3.1 Code pattern of the module P described in the text.

In figure 5.3.1 we show a code pattern which the module P mentioned earlier could possess. The construct:

```
associate V:C;
```

means that the carrier parameter list C (a variable here) is to be associated with the dummy parameter list D_i of the module Q_i contained in V. It follows then, that by means of

this construct also the root parameter list R which corresponds to C is associated to the non-root parameter list N_i which corresponds to D_i.

In section 5.2 we have shown that, in order to associate root parameters with non-root parameters, parameter names are required [named association rule, definition 5.2.1]. Furthermore, only those parameters bearing the same parameter name are associated. Now, the root parameter list R is defined without knowledge of the module Q_i contained in V when R is associated with N_i. There are then two alternatives:

i- parameters which serve for the same purpose have equal parameter names in all non-root (dummy) parameter lists N_i. This solution is not satisfactory since it restricts the parameter naming freedom and, possibly, forbids a module M from being a member of the module sets Q and Q′ of two different module typed variables V and V′.

ii- define a mapping which maps the parameter name of the root parameter list R onto a parameter name in the non-root parameter list N_i. It should be clear that this mapping must be programmer defined, since it has to be purpose-wise correct.

Defn. 5.3.2 Name equivalencing is an operation which maps the parameter names of the root parameter list related to a module typed variable V to parameter names of the non-root parameter list N_i of the module Q_i contained in V when association occurs.

Name equivalencing could map one root parameter name onto zero or more non-root parameter names. However, at

most one root parameter name may be mapped onto a given non-root parameter name, since, otherwise, the association would be ambiguous in the general case.

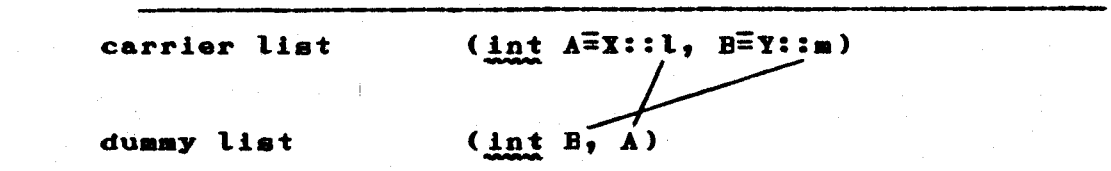


Figure 5.3.2 Association with name equivalence.

In figure 5.3.2 we show an example of how parameters could be associated by means of equivalenced parameter names. The carrier parameter list defines the root parameters "X" and "Y". These parameter names are equivalenced respectively to "A" and "B". This allows then to associate the root parameter "X" with the non-root parameter "A" and, similarly "Y" with "B". Due to this root and non-root parameter list association, the carrier parameter "l" is associated with the dummy parameter "A" and, similarly, "m" with "B". Recall that dummy and non-root parameters usually bear the same textual name.

We have now that, whenever a root parameter list R is to be associated with a non-root parameter list variable N_i, a name equivalence must be established first. Let us examine now whether this could be accomplished when assigning a module typed value to the variable V.

In order to associate a carrier parameter list C with a dummy parameter list D_i, the list C is linked to the identifier of D_i in the associate construct. Usually the identifier of D_i is the module name Q_i of the module defining D_i. In the case of module typed variables, however, this identifier is the (textual) name of V. That is, for the purpose of parameter association, C is linked

to V regardless of the contents of V. Recall that, in order to associate C and D_i, it suffices to examine how the association between the corresponding root parameter list R and non-root parameter list N_i is accomplished.

Defn. 5.3.3 Let V be a module typed variable. A <u>template parameter list</u> T is a parameter list which is defined together with V and allows the name equivalencing between a root parameter list R and the non-root parameter list N_i of module Q_i contained in V to be broken into two name equivalencings, one between R and T and the other one between T and N_i.

In the usual case the name equivalence between R and T is the identity equivalence. Thus, in this case, the name equivalence between R and T may be implied.

It should be clear that, with the aid of template parameter lists, we are able to associate the carrier parameter list C with the dummy parameter list D_i without having to know the contents of V. This follows from the fact that the template parameter list defines a "standard" carrier parameter list CT which is associated with D_i. Furthermore, it defines also a "standard" dummy parameter list DT which is to be associated with C. Suppose now that V does not define the template parameter list T. It follows then, that we must know the contents of V in order to associate C with D_i, since not necessarily is the name equivalence between C and D_i the same for all D_i in the module set Q. Formalizing we have:

Lemma 5.3.4 Let D_i be the current dummy parameter list identified by the module typed variable V, where D_i is to be associated with the carrier parameter list C linked to V. The named association rule can

be implemented in the general case, iff V defines a template parameter list T such that C associates with T which, in turn, associates with D_i.

The textual occurrence of a carrier parameter list defines in fact a carrier and a root parameter list typed value, i.e. constant. Within a given module there could be several such constants all linked to the same module typed variable V. Let $C_1, C_2, \ldots, C_n$ be the collection of these parameter list typed constants. If the name equivalence between all C_j and the template parameter list T defined together with V is the identity equivalence relation, it follows that the parameter names of the parameters which serve for the same purpose in all C_j has to be equal. Since all C_j are defined within the same module this is not difficult to enforce. Suppose now that the carrier parameter list C linked to V is in fact the contents of a carrier parameter list typed variable. We must now define a name equivalence from the contents C_j of C to T in such a way that the association from C_j to D_i is purpose-wise correct. Since C ignores the contents of V it follows:

Lemma 5.3.5 Let T be the template parameter list relative to the module typed variable V. The association of a carrier parameter list typed value C_j with the dummy parameter list D_i of the current contents Q_i of V is purpose-wise correct and independs from the contents of V, iff the purpose of each template parameter t within T is well understood.

Since T is defined together with V (possibly dynamically), T is known when Q_i is assigned to V. It follows then, that we can establish a name equivalence between the parameter lists T and N_i when Q_i is assigned to

V. It should be clear that T could contain less parameters than N_i and, consequently, some of the parameters in N_i would remain non-associated if the name equivalencing from R to N_i is necessarily via T. We will deal with this case later in this section.

Let W be a module typed variable distinct from V. Let U be the template parameter list related to W. Suppose now that we assign V to W, e.g. W:=V. If in doing so a name equivalence between T and U is established, we have that there is a name equivalence between U and N_i, where N_i is the non-root parameter list of the module Q_i contained in V when the assignment is performed. We have then:

Lemma 5.3.6 Let V be a module typed variable and T the template parameter list related to V. A partial name equivalence between T and the non-root parameter list of the module Q_i contained in V can always be established when assiging a module typed value to V.

In order to maintain the sequentially multityped environment, the conditions of lemma 5.1.1 must be satisfied. It follows then, that we must perform type checking when associating parameters. In order to decrease run time effort, this checking should be performed statically[Hab2]. Suppose now that the template parameters are not typed. Thus, when associating the template parameter list T with the dummy parameter list D_i, type checking will have to be performed, since, in the general case, it is not known whether the actual carrier parameter types in T agree with the corresponding dummy parameter types in D_i.

Lemma 5.3.7 Static type checking of parameters in the presence of module typed variables is possible only if template parameters are explicitly typed.

Suppose now that a template parameter t is of a union type τ_t, where $\tau_t=\{\tau_{t1},\tau_{t2},\ldots,\tau_{tm}\}$ is the set of effective types of τ_t. Let d be the corresponding dummy parameter. Furthermore, let $\tau_d=\{\tau_{d1},\tau_{d2},\ldots,\tau_{dn}\}$ be the set of effective types of the type τ_d of parameter d. Now if τ_d does not contain τ_t it follows immediately that a run time type checking is required in the general case when associating t and d. We have then:

Lemma 5.3.8 Even when template parameters are explicitly typed, static type checking is not necessarily possible in all cases.

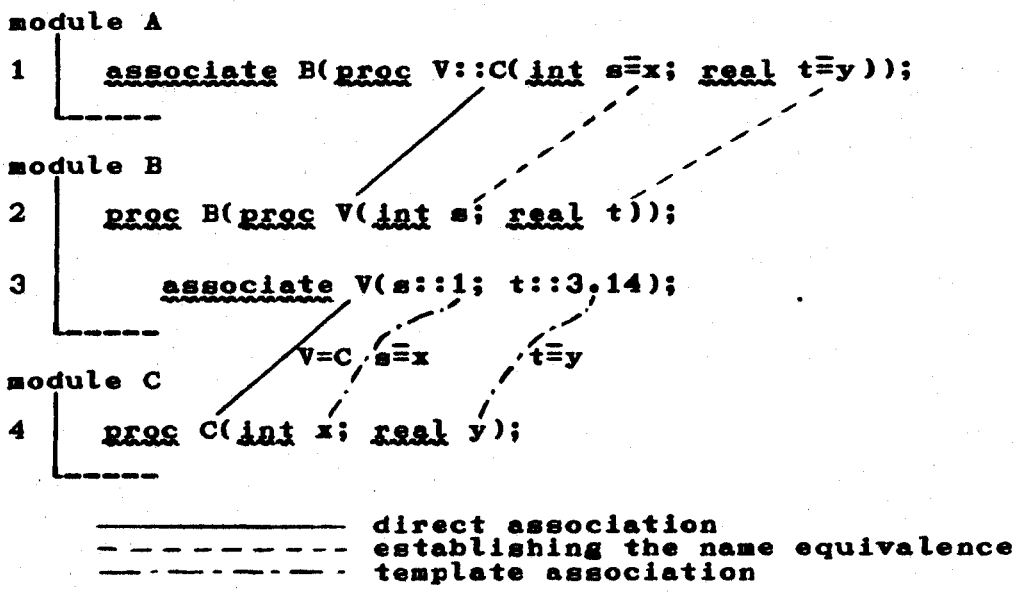


Figure 5.3.3 Example of parameter association in the presence of a module typed variable.

In figure 5.3.3 we show an example of how the parameter association mechanisms defined in this section work.

The module typed variable V and its corresponding template parameter list are defined in statement 2. Statement 1 corresponds to an assignment to V and also establishes the name equivalence between the dummy parameter list D of C and the template parameter list T of V. Consequently, the name equivalence between the non-root parameter list N of C and the template parameter list T of V is established.

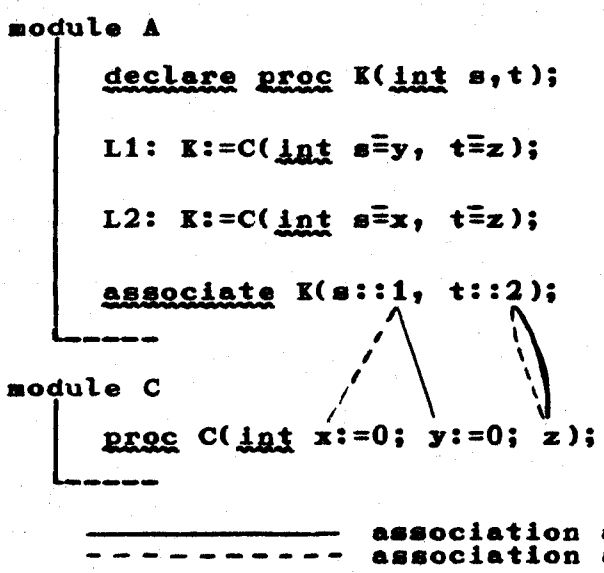


Figure 5.3.4 Distributivity of parameter associations due to different name equivalence definitions.

In figure 5.3.4 we show how different associations can be achieved by means of different definitions of the name equivalence. Observe that this distributivity could not be accomplished by means of the positional association rule. Observe also that the dummy parameters "x" and "y" define default parameter values, i.e. "x:=0" and "y:=0". It follows then, that the elaboration of the module C will never fail due to undefined parameter values, even if some of these dummy parameters remain non-associated.

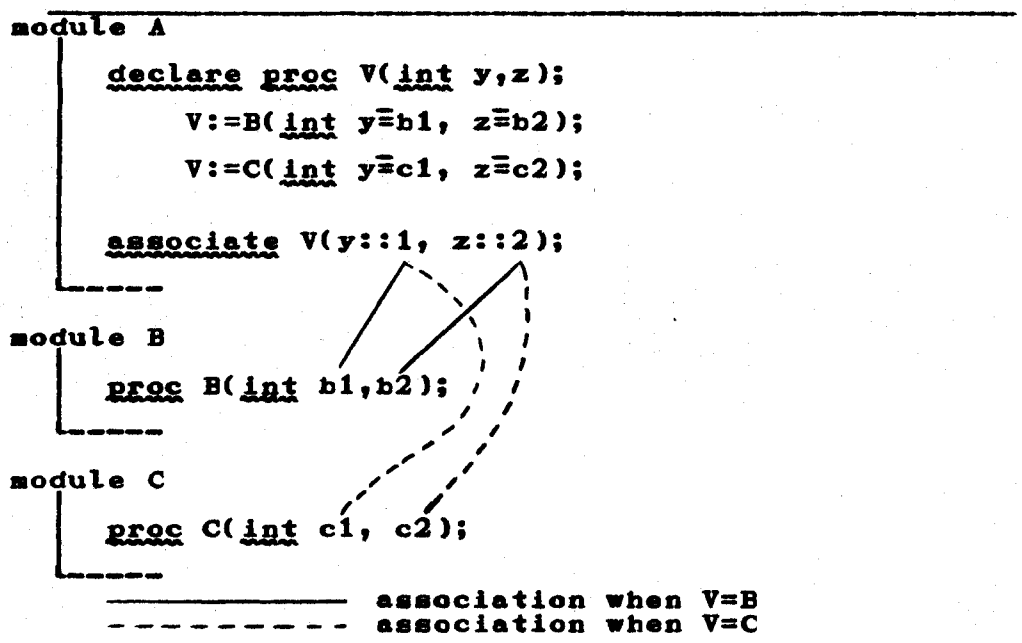


Figure 5.3.5 Example of associations relative to different non-root parameter lists.

In figure 5.3.5 we show an example where the module typed variable V could effectively contain one of several module typed values. Observe that the name equivalence is defined whenever a module typed value is assigned to V.

We have assumed so far that the template parameter list T defines sufficient parameters in order to allow the partitioning of the name equivalence as stated in definition 5.3.3. Not always is this the case though. Consider, for instance, following problem: there is a complicated procedure P [e.g. a plotting procedure] which accepts as parameter another procedure V [e.g. the function to be plotted], where this procedure typed variable V is defined with only one template parameter.

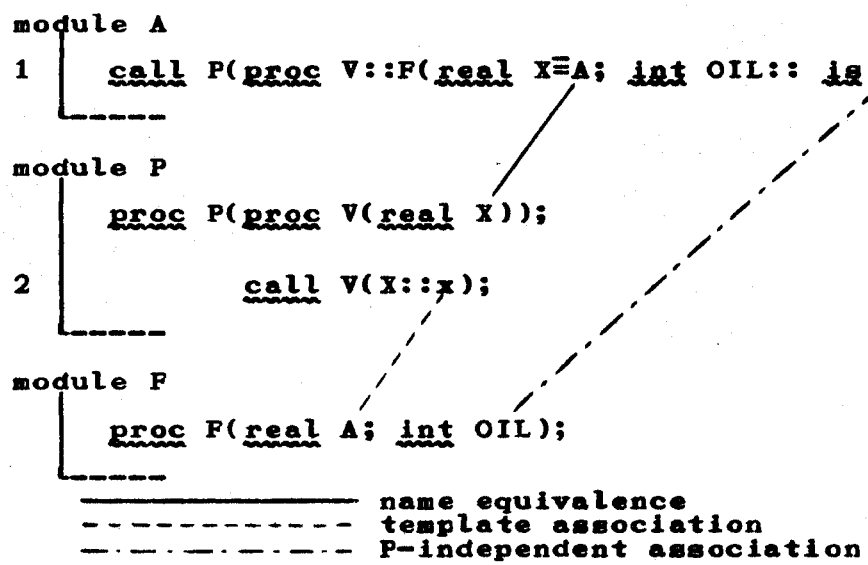
If the actual funtion F to be plotted requires at most one parameter, the procedure P is perfectly adequate. Suppose now that F requires more than one parameter, however,

except for one of these parameters, all others remain invariant for a given plot. It follows then, that P could still be used to plot F. For example, F could be a function which determines the viscosity of an oil at some temperature t. The plot to be drawn would then be a viscosity chart of several oils with respect to the temperature.

The first solution we will examine is frequently adopted in languages such as FORTRAN IV and ALGOL60. Basically this solution consits in prefixing the procedure F by a single parameter procedure F′, where F′ receives the parameter computed by P and combines this parameter with the other (invariant) parameters required by F. Usually the invariant parameters incorporated by F′ are obtained by means of local/global parameter associations. This solution is not satisfactory, since it may create naming problems with respect to the global parameter names and, also, it increases the complexity of the program.

The second solution we will examine is based on the fact that association of dummy and carrier parameters can be performed piecemeal. It should be clear that, by means of the operation association, piecemeal associating dummy and carrier parameters could be achieved. For example, the dummy parameter list implementation could define a table where the values transmitted, e.g. references, are placed. Thus whenever association is performed this table is updated. Finally, due to the use of the named association rule [definition 5.2.1], association of dummy and carrier parameters can occur in any order.

In figure 5.3.6 we show an example of how piecemeal associating dummy and carrier parameters could be achieved. The verb is denotes that the dummy parameter "OIL" of



5.3.6 Example of piecemeal associating carrier and dummy parameters.

procedure "F" is to be associated immediately with the carrier parameter "I". Thus after elaboration of statement 1, the dummy parameter "OIL" of procedure "F" is associated to the carrier parameter I, whereas the dummy parameter "A" still remains to be associated. This parameter "A" will eventually be associated when control reaches statement 2.

5.4 Miscellaneous Topics Regarding Information Interchange.

In this section we will study several independent aspects of information interchange. We have chosen to collect these information interchange aspects into one separate section, since none justifies a section of its own, nor does any of them fit easily into some previous section.

Consider the information interchange mechanism of IPL V. IPL V defines several global data spaces, i.e. cells. A cell is a pushdown stack and it is used to preserve, i.e.

push, and to restore, i.e. pop, the information contained in the top of this stack. The information in the top of a cell is the information available for processing.

Observe that a storage mechanism such as cell can be implemented by means of a type descriptor similar to the type descriptor "stack" introduced in section 3.3 [figure 3.3.1]. Since we impose no restrictions on types of parameters, it follows that the language constructs which we have introduced so far are perfectly suited to implement the IPL V cell.

One of the interesting properties of information interchange via cells is that the information in a cell is kept until it is eventually "consumed". Thus, two modules M and N may interchange information even when neither knows the other and/or the flow of control route which leads from one to the other. That is, M (or N) "services" data as it becomes available in a rather flexible way. Observe that this allows a module M to post information about some "exceptional" condition. This information may then be used by another module N, or may be left unchanged altogether. For example, two modules M1 and M2 could have been designed to solve the same problem. Another module N could have been designed to decide which of these two modules M1 and M2 is the fastest. Thus both modules post their execution time, however, once the decision has been made, the module N may be disabled without that any change to either M1 or M2 has to be made. It should be clear that problems with regard to timing may occur, we will not examine these problems though.

Observe that we could define cells to possess a behaviour which is different from the stack behaviour. For

example we could define cells to be queues. This allows us then to establish a "pipe-lined" processing environment.

Consider now the Jensen device[Ruti]. We have here a procedure typed parameter, i.e. a call by name thunk. Furthermore, there are other parameters, possibly global, which are changed by the called procedure and are also parameters to the thunk. Thus, if such parameters are changed (by side effect), the thunk may produce different values for successive activations.

We will establish now a notation aimed at reducing ambiguities in the subsequent text. Let P be a procedure which defines a carrier parameter list C, where C contains a module (thunk) typed parameter t. Let T be the actual value, i.e. module, bound to t when association occurs. Observe that T is usually defined within C itself. Let F be the module containing the dummy parameter list D which is to be associated with C. Thus D contains a module typed parameter t'. Let $S=\{s_1,s_2,\ldots,s_n\}$ be the set of parameters in C which are associated with both D and the dummy parameter list DT of the module T. That is, if F changes by side effect any of the parameters s_i in S, T will possibly produce a value which is different from the value produced before this change took place.

Usually the dummy parameter t' has the aspect of a variable, i.e. it appears as if it were a parameterless procedure. In section 5.3 we have introduced the is construct. By means of this construct we could associate the parameters of DT and D at the same time. Observe that T will still be sensible to changes of these parameters if the association is implicit, or if filling is performed whenever the elaboration of T is begun.

```
module P
    call F(proc T::(int L:: is I) <procedure body
                                   using L>; int S::I);

module F
    proc F(proc T; int S);
```

Figure 5.4.1 Using the is construct to implement Jensen's device.

In figure 5.4.1 we show how Jensen's device could be implemented using the is construct defined in section 5.3. Observe that this implementation follows exactly the ALGOL60 implementation except for the fact that here we are using the named association rule rather than the positional association rule of ALGOL60. A major drawback of this solution is that the parameters in the set S must be mentioned twice, first in order to associate DT (is part) and then in order to associate with D. Besides being annoying, this may cause errors to go by unnoticed and, also, makes it more difficult to understand the program.

```
module P
    call F(proc T:: depending on(int L≡S::I)
                                   <procedure body using L>);

module F
    proc F(proc T; int S);
```

Figure 5.4.2 Making Jensen's device explicit.

This suggests then an alternate approach to this problem. In fact what we want is to turn explicit that a

given carrier parameter is to be associated simultaneously with two (or more) dummy parameters each of which is defined in a different dummy parameter list. That is, we want to put the set S into evidence. In figure 5.4.2 we show how this could be achieved. The verb depending_on means that the parameters occuring in the depending_on list are to be associated with both the dummy parameter lists DT of the thunk T (defined within C), and the dummy parameter list D of module F.

The depending_on construct could be extended in order to allow more than two dummy parameter lists to be associated at once. For example, the name equivalence defined within depending_on could define the qualification <module.dummy list.parameter name>. It is possible then to associate the parameters in S to several modules, as well as to modules which are not defined within the carrier parameter list C.

When developing a software package, e.g. a plotting package, frequently such a package is a collection of modules which are logically interdependent. That is, these modules possess their own information interchange properties which, usually, are to be hidden from the exterior. Such collections of modules could be viewed as a type which we will call group.

In many cases we do not know when designing a module whether it will be a member of some group or not. Thus any module, in particular library modules, must be able to be included into a group. Furthermore, some of the modules included into some group may well be a group themselves.

Modules in a group frequently establish intercommunication areas which are global to all the modules in the

group. These areas are to be hidden from the exterior though. Observe that exactly the same occurs with respect to type descriptors and generator functions. Thus the mechanisms introduced to implement type descriptors and generator functions will also be used to implement groups. Group instances will then be created by means of a possibly dynamic declaration. As has already been shown, by means of such a declaration the internal global areas can be allocated and initialized according to some externally provided parameter. Furthermore, there are no restrictions of whether a module within a group is a coroutine and/or is recursive. Finally specific parameters can be passed to individual modules within a group without any restriction whatever.

It remains now to define how modules are grouped. For each module the required interface information is defined. In particular the non-root parameter lists are defined. It is possible then to perform carrier and dummy parameter associations when grouping modules, thus forcing certain parameters to be bound to group information interfacing areas. We may conclude then, that the implementation of a grouping construct could be achieved quite easily by means of the constructs developed so far.

6. Exception Handling.

In this chapter we will study the handling of exceptions. There are several reasons why such a study is important:

a- programs might be interrupt driven. For example, interactive systems, e.g. time-sharing systems, usually allow executing programs to be preempted and later to be resumed by means of user interaction.

b- some operations may fail and this failure may direct further action, where such a failure is not to be regarded as an error. For example, when reading records from a sequential file, e.g. tape file, a read request may fail due to the file having been exausted. In this case a summary submodule is frequently started.

c- a given module M could define several submodules, each of which attempts to solve the same problem in a different way, e.g. using different algorithms and/or starting values. By means of successive trials of these submodules and/or starting values, the module M could eventually produce the desired result.

d- machine failures could cause program elaboration failures. Such failures are non-avoidable since every machine possesses a (usually very slim) probability of malfunction. Notice that such failures occur frequently in input/output handling modules and appropriate safeguards are incorporated into these modules.

e- the program itself may contain errors and thus reach abnormal states. Ideally programs should be proved

correct[Dij4,Flo1,Hoa2]. Pragmatically, though, this ideal cannot be achieved when using the tools currently available[Hor1]. Thus even when programs have been "proved" correct, they may still be incorrect[Sch4] possibly due to incorrect proofs, misunderstanding of the program environment, unforeseen conditions etc.

From items (a) through (c) it follows that we may desire to use exceptions as a tool for solving a given problem. From items (d) and (e) it follows that we must be able to cope with unexpected exceptions in order to prevent serious damages to the program (or system) and to produce information which, hopefully, aids diagnose the fault.

We will study in this chapter how exception conditions can be defined so as to allow user defined conditions and parameters to exception handlers. Since exception handlers can be equivalenced to data flow driven modules, as has been shown in chapter 5, we are in fact studying here the tools necessary to implement data flow driven modules.

The point where an exception is detected is not necessarily within the module instance which defines the corresponding exception handler. We must examine then how a detected exception may be passed from module instance to module instance in order to be serviced eventually. We will examine also the ways in which control can be given back to the module instance where the exception was detected.

Due to the parallel or quasi-parallel nature of exception handlers, there are several timing problems which must be examined. Our study will concentrate in determining the different timing problems and how they could be overcome.

However, we will not study how deadlocks could be detected and/or prevented.

This chapter is subdivided into 4 sections. In section 6.1 we present a survey of the exception handling facilities available in present day programming languages. We expect to be able to abstract a set of primitive constructs required for exception handling from this survey.

In section 6.2 we study how exception handlers could be defined so as to allow both user defined conditions and dynamically associated parameters to be transmitted to the user defined exception handler. In this section we study also the flow of information between the handler and the other modules, possibly handlers.

In section 6.3 we examine the forms in which the exception handler may interact with the module where the exception was detected and, also, with other modules which aid in the handling of the exception. That is, we study the flow of control between the handler and other modules, possibly handlers.

Finally, in section 6.4 we will examine the problems which arise due to the existence of non-null time intervals between detection and handling.

6.1 Exception Handling in Existing Programming Languages.

In this section we will survey the exception handling capabilities of several existing programming languages. The object of this survey is to discuss the various mechanisms which have been developed so far, with the goal of deter-

mining whether a common basis for all of these mechanisms could be defined.

We will study following languages: PL/1[IBM2,IBM3], SPITBOL[Dew1], SNOBOL4[Gri6], IPL V[New1], ALTRAN[Bro4], APL\360[Pak1]. It should be noted that we are not attempting to survey all programming languages which provide some sort of exception handling.

We will define now the terminology which we will use henceforth. An exception is a detected instance of an exception condition. For example, overflow is an exception condition, whereas the occurrence of overflow is an exception. It should be clear then, that exception conditions describe exceptions in the sense that these conditions define how an exception is to be detected. A module which has been designed to handle exceptions will be called exception handler, or handler for short.

The key points of interest in this survey are:

a- the different classes of exception conditions defined by the language processor. For example, we will examine whether the language processor allows external interrupts, run time error detection, e.g. division by zero etc.

b- whether the language allows users to define their own exception conditions and then, whether the language processor performs automatic detection of these user defined conditions.

c- the system actions which are performed when exceptions are detected, and whether these actions could be redefined by the user.

d- the information made available to user defined exception handlers.

e- the ways in which control may be passed from the module instance where the exception has been detected to the corresponding exception handler and back.

f- whether a given exception could be passed from handler to handler in order to be serviced, and/or whether detection of exceptions has to be followed immediately by the elaboration of a handler.

PL/1

PL/1 defines several exception conditions which are identified by language defined textual names, e.g. ERROR, ENDFILE etc. Language defined conditions may be induced explicitly by program request [SIGNAL]. For each of these language defined exception conditions a standard language processor action is defined. These actions vary considerably from condition to condition. PL/1 allows users also to define their own exception conditions [CONDITION condition]. However there are no provisions for automatic detection of user defined exceptions in PL/1.

Exception handlers are identified by the same textual name as that which identifies the condition. These textual names are always globally defined for the entire program [EXTERNAL]. There may be several exception handlers all identified by the same textual name. Handlers are dynamically defined within module instances [ON statement]. This definition may refer to a user defined module [on-unit] or it could refer to a system defined module [SYSTEM]. Observe that a handler corresponds to a program module which is to be elaborated when the exception is detected. Creation and

activation of an elaboratable module instance of the handler occurs when and only when the corresponding exception is detected. It follows from this activation criterion, that exception handlers could cause infinite recursion loops if they themselves detect the exception they handle.

A new handler is bound to the exception condition identifying textual name only if no handler instance is already associated with the module instance defining this handler. Otherwise, the binding associated with the module instance is changed from the old handler to the new handler. We may conclude then, that the textual names which identify exception handlers are of type "stack of module".

The module chosen to be elaborated when an exception is detected is the one which is defined at the top of this stack. This stack is never empty and the bottom element is the system action module corresponding to this exception condition. The top element of the stack is automatically deleted if the module instance within which it had been defined is terminated. It may also be deleted under program control [REVERT]. Notice that this may cause interference with other modules.

The textual scope of textual names occuring within a given exception handler obeys the usual block structured scope rules. The textual names occuring within handlers are bound to the environment within which the module instance defining the handler exists. That is, the handler's global parameters are bound to the local parameter instances which are dynamically active at the moment of handler definition. It follows then, that the run time manager of PL/1 must be able to reconstruct temporarily some previous environment when the exception is detected in order to allow the

handler to elaborate. Furthermore, the environment at the moment of detection must be reestablished when the handler finishes elaboration.

Exception handlers communicate with other modules by means of local/global parameters. There are also several built in functions which allow obtaining more precise information regarding a given exception [ONCHAR, ONCODE]. There is no provision for actual/formal parameters though. In many cases this makes it quite difficult to obtain the precise information with regard to a given exception. For example, when overflow occurs it is difficult to determine the statement, the operation and the values which caused the overflow.

Once the user defined handler ceases elaboration, some further language processor action takes place. This action determines whether elaboration should be terminated [ERROR, FINISH], whether the operation which detected the exception should be retried [CONVERSION] or, finally, whether the elaboration of the preempted module instance should resume at the next following operation [ENDFILE, CONDITION]. It is also possible to terminate the elaboration of the exception handler by forcing elaboration to resume at some user defined point in the program [goto out of handler].

SPITBOL

The only exception condition which exists in SPITBOL is the run time error. SPITBOL does not provide means which allow users to define their own exception conditions. Errors are detected automatically and cannot be induced directly by means of an explicit command. Errors are con-

sumable objects and the programmer can determine dynamically how many non fatal errors will be allowed [&ERRLIMIT].

Statements in SPITBOL (i.e. SNOBOL4) may either succeed or fail. Failure may be due to several reasons such as failure to match a string, false as the result of a relation, run time error etc. Each statement defines explicitly the statement where control is to be sent to in the case of success or failure. An undefined success and/or failure goto field means that the next statement in succession is to be elaborated if, respectively, the statement succeeds or fails.

The usual system action for non-fatal run time detected errors is to branch through the statement failure goto field. This system action may be overridden by a user defined action [SETEXIT function]. This is accomplished by defining a label to which control is to be directed in the case an error is detected. Notice that SPITBOL does not differentiate between the several detectable non fatal run time errors. It is then the programmer's responsibility to determine the error cause when designing the error handler [&ERRTYPE].

All information defined within the environment at the moment of error detection is available to the handler. Furthermore, the user may change this information freely. After elaboration of the handler body, the user defined error handler may choose to terminate the program [goto ABORT], to remain within the error handler or to resume the elaboration of the program at the statement defined by the failure goto field of the offending statement [goto CONTINUE].

Error handler entries (labels) are "consumable" objects. That is, once an error has occured and control has been given to a user defined error handler, any further error will be handled according to system defined actions, unless the user defines a new error handler entry. Notice that in the case of multiple errors, this may lead to elaborations which are difficult to understand, since all errors following the first detected error may pass unnoticed, due to the system action for detected non fatal run-time errors being to branch through the failure goto field of the offending statement.

SNOBOL4

In SNOBOL4 we are able to define exception conditions by means of the program trace feature. This follows from the fact that the user may define a function which is to be elaborated whenever the data space bound to a given textual name is changed. For example, error handlers could be defined as functions which are elaborated whenever the contents of the system defined error counter [&ERRLIMIT] is changed.

Observe that within our type definition and checking framework we could enforce tracing by defining O-emulating functions within a given type descriptor [see section 4.4]. Thus, by means of type descriptor transmission, the existence of these tracing function can be propagated to all modules which use this particular type.

Concluding the survey of SNOBOL4's exception handling capabilities, observe that we could associate a same user defined trace function to several different variables. Now, if we precede the body of this function by an "information

presence" predicate such as defined in section 5.1, the activation of this module could effectively be dependent on quite complex conditions. Furthermore, the testing of the condition could be considered as being automatically provided by the language processor, since every change to one of the associated variables will cause an automatic reelaboration of the "information presence" predicate. User defined trace functions are called with two actual/formal parameters. One of these parameters identifies the variable being traced, it is then quite easy to determine the origin of the trace function call.

IPL-V

IPL-V distinguishes two basic exception conditions: the external interrupt and the run time error [error trap]. External interrupts are detected and serviced at program defined monitor points [Q=3]. The external interrupt service module must be defined by the user. If it has not been defined when an external interrupt is sensed, the corresponding action is a no-operation [J0].

Errors are detected by the language processor and/or induced by program action [J170]. Associated with each error there is an error code which could be user defined. Whenever an error is detected or induced, a language defined associative list [W26] is searched for the error code provided. If the error code is found, the associated entry names the error handler to be elaborated. If the original error code is not found in this list, the list is searched again, this time for a standard language defined error code [internal zero]. If found, the associated entry names the error handler to be elaborated. If not found, the error condition is simply ignored.

Observe that the above mentioned associative list can be modified by the user. Thus, error codes and their associated module names could be included and/or deleted from this list. Furthermore, the module names associated with a given error code could be changed by program action. It should be clear from this discussion that IPL-V provides explicit names (codes) for the exception conditions and their corresponding handlers.

Since all modules communicate through globally defined cells (stacks), all information available at the moment of error detection is also available to the error handler. Furthermore, the statement counter and the error code are made available as additional information [W27,W28]. Once the error handler terminates, or if it has not been defined, this additional information is discarded.

We have already mentioned in section 4.4 that exceptions may be user detected and the corresponding descriptive information placed into some cells defined for that purpose by program convention. The appropriate action could then be triggered at some later instant, e.g. due to an external interrupt, or due to the elaboration of a clean-up procedure. Notice that this allows a given exception to be left pending, possibly to never be serviced. Thus in IPL-V exception detection does not have to be followed immediately by the elaboration of the corresponding handler.

ALTRAN

The only kind of exception condition available in ALTRAN is the run time error. Run time errors may be induced by program request [FRETURN]. Furthermore, the

error code associated with a user requested error return may be user defined.

The normal system action when an error is detected is to place the offending module into the error state. After this, this module is searched for a user defined error handler [label SYSERR not empty]. If the handler is defined, elaboration resumes with this handler being elaborated. Otherwise an error return to the calling module is performed and this calling module is also placed in the error state. After this the aforementioned process is repeated. Thus, if there are no user defined error handlers, eventually the main procedure is terminated by means of an error return and, consequently, the elaboration of the program is terminated.

When handling an error, module instances may be created by conventional module calls. It would be possible then, that a given module instance performs an error return to a module instance which is already in the error state. However, double errors are fatal errors in ALTRAN, thus such a return would cause an immediate termination of the program. Similarly if an error is detected when elaborating a module instance which is already in the error state, elaboration of the program ceases immediately.

The error handler H may decide that the program has recovered from the error [OK( )] or, by means of an error return, may pass the error to the module instance M which invoked the module instance M′, where M′ contains the exception handler H. Thus, ALTRAN allows exception to be passed from one exception handler to another exception handler.

Several parameters are made available to the error handler by means of library procedures [HERRNO, LERRNO]. The internal data of module instances which are terminated due to a non serviced error is placed in a special language defined list [dumping list]. This information is not directly available to the error handler though, and is usually transmitted to some file, e.g. print file [SNAP].

APL\360

There are several exception conditions in APL\360, e.g. run time detected errors, programmer defined break-points [stop control], attention key pressed at the user's terminal etc.

Exception handlers are not invoked automatically by the language processor. Rather the elaboration of the program ceases and the terminal is placed in the execution mode. The user is then free to display and/or change all data spaces defined within the environment of the preempted module instance. It should be clear that this examination and display of data spaces could be carried out by a user defined "program", i.e. function. However, this function can only be activated by explicit user interaction.

After the above described investigation, the user may choose to resume elaboration of the preempted module instance [goto existing statement number], to terminate the preempted module instance [goto non existing statement number] or to terminate this particular program [goto null statement number].

Resumption of module instances is accomplished by means of a so called state indicator. This state indicator

keeps the dynamic history of the running program. It is thus the run time environment stack and it contains:

i- the module instances and the corresponding statement numbers of the statements which were being elaborated (or about to be elaborated) when the exception was detected.

ii- the internal environment of each of the module instances, i.e. the bindings of the local variables of the module instances.

Notice that a resumption operation is nothing more than a traversal of the set of handler instances in the reverse order in which they were activated. Observe that this traversal occurs beyond the program's control in APL\360.

Summary

From the preceding survey we may conclude that a generalized exception handling facility should be designed in such a way that:

a- exception handlers should be treated as module typed variables (PL/1, IPL-V, SNOBOL4).

b- the textual name of a given exception handler could identify a family of handlers and also a decision procedure which chooses one of these modules when the corresponding exception occurs (PL/1).

c- changing the relevant exception handler should be an excecutable operation, rather than being textually determined (SNOBOL4, SPITBOL, APL, IPL-V, PL/1).

d- users should be able to define their own exception conditions and/or handlers. Furthermore, the language

processor should provide for the automatic detection of user defined exceptions (SNOBOL4 and generalizing PL/1, IPL-V).

e- exception handlers should be consumable objects in the sense that only a limited number of module instances of the handler could exist simultaneously (generalizing ALTRAN, SPITBOL).

f- exception handlers should have the ability to examine an exception and either handle it or pass it to some other exception handler (ALTRAN, IPL-V).

g- exception handlers should allow information to be interchanged by means of dynamically associated parameters (generalizing SNOBOL4).

h- exception handlers should have access to the data at the time of the exception (APL, ALTRAN, IPL-V).

i- the detection of exceptions should not necessarily be followed immediately by an exception handler elaboration (IPL-V).

It should be noted here that we are not examining the merits of above proposals in this section.

6.2 Exception Descriptors.

In this section we will study how exception conditions are defined. It will be shown that exception descriptors include the definition of the detection operation and the information to be interchanged with the handler.

We will concentrate on the problems relative to exception detection, and on linguistic and information flow

aspects. We will not consider timing problems in this section. These problems will be dealt with in section 6.4.

An exception E is information and, thus, is a value of some type T_E, e.g. boolean. The type set T_E must distinguish at least one value v_0 in T_E and attach to it the meaning that E has not been detected, since, otherwise, it would be impossible to determine whether E should be serviced or not by some exception handler H. In the general case only the user is capable of determining whether a given exception E has been serviced. It follows then, that the user must have the power of placing v_0 into the data space of E. For example, when exceptions are detected by hardware interrupts, the user must be able to reset such an interrupt signal.

Exceptions are detected when a given information, say the detection information, satisfies a given condition, say the detection condition. For example:

a- tape end of file is usually detected when a "tape mark" (detection condition) is read from the tape (detection information).

b- in some machines, overflow is detected by testing whether the "overflow toggle" (detection information) is "on" (detection condition).

In some cases the detection is performed by hardware, e.g. (a) above. This hardware detection of exceptions may cause the elaboration of the current module to be suspended and another module, e.g. the interrupt handler prologue, to be activated. Observe that this is similar to a module activation (procedure call), however, neither is there an explicit activation operation, nor is this activation

restricted to occur only at one of several predefined points within the preempted program.

The code point where the exception is detected will be called detection point. As noted above, the detection point and the code being elaborated do not necessarily possess any logical connection whatever. Observe that the detection point is time dependent and thus is always dynamically defined. Of course, it could be statically known that, whenever the elaboration traces a certain code point, detection will occur. This is the case, for example, with respect to PL/1 SIGNAL statements and O-emulating functions [see section 4.4]. In this case the handler could be activated by means of an implicitly included procedure call or macro expansion. Even when the detection point is explicitly related to the code, e.g. a division which causes an overflow interrupt, this detection point does not necessarily correspond to a well defined textual point in the users program. It follows then, that a language construct must be provided to make the detection point explicit whenever necessary.

The detection information is kept within one or more data spaces. It follows then, that, if we want to perform detection as early as possible, all changes to these data spaces must be followed immediately by an evaluation of the detection condition. We have then:

Lemma 6.2.1 Immediate detection of exceptions is possible only if all operations which change a data space carrying part or all of the detection information also verify whether the detection condition is met.

Lemma 6.2.1 shows that, in order to allow user defined detection conditions to be verified immediately, we must be

able to extend, i.e. emulate, all data space changing operations, so that these operations perform the required detection condition evaluation. This implies that we must know these operations and that we are allowed to extend them. We will call these extensions detection extensions. Notice that detection conditions could be simulated by procedure calls, thus making lemma 6.2.1 trivially true, since the "call" stands for the condition becoming true.

A data space α could be changed due to the existence of operations on α which invoke user defined procedures. It follows then, that, in the general case, it is undecidable to know whether a given data space is possibly changed by a given operation. Thus every operation making use of a user defined procedure must be assumed as changing the data space. This may result in an execution cost increase due to the repeated and unnecessary evaluation of the detection extension. It follows then, that, for practical purposes, a construct should be provided which allows to define an operation as being not data space changing. It should be clear though, that it is the programmer's responsibility to enforce that the operation is de facto not changing.

```
when (overflow): ...
        - - - - -
integer I, J;
when (I,J) begin I<J end: ...
```

Figure 6.2.1 Examples of detection information and conditions.

In figure 6.2.1 we show two examples of detection information and detection conditions. We will use the keyword when to denote the definition of an exception descriptor. In the first example, detection is performed by

an overflow signal and should be self-evident. Notice that in this example the exception name is explicit, i.e. overflow.

In the second example we show a user defined detection condition. In this case we must monitor all changes to I and J. Whenever I becomes less than J, the exception is effectively detected. Observe that in fact we are dealing with several exceptions in this example. This follows from the fact that the detection condition "I<J" is tested within the exception handler of the "change completed" exception signalled whenever I or J are changed. Notice that in this example the exception name is implicit.

```
type test_integer(condition handler)=
   extend integer with
      emulate store=signal(handler);
         - - - - -
test_integer(condition handler:: I_J) I, J;
when(I_J): test=if I<J then signal(I_lt_J); fi;
when(I_lt_J): ...
```

Figure 6.2.2 Manual creation of detection extensions.

In figure 6.2.2 we show how the second example of figure 6.2.1 could have been extended manually, so as to include the required detection extension. Observe that exception extensions, such as shown in figure 6.2.2 and implied in figure 6.2.1, are implementations of conditional critical sections as defined by Hansen[Han3]. Notice also that the exception names are all explicit in this example, i.e. "I_J" and "I_lt_J".

Notice that we are assuming that the system is able to generate automatically the detection extensions. It should be clear that automatic generation of detection extensions is not necessary since programmers themselves could provide them as shown in figure 6.2.2. Manual extension has several drawbacks though. It leads to a hard to understand program, it increases the probability of incorrect code and, finally, it makes it more difficult for the language processor to perform the change monitoring in an efficient way.

It should be clear that a detection extension of a type τ usually defines a new type τ' such that identity conversions are defined between τ and τ'. Thus, data spaces of type τ could, in principle, be retyped freely to τ' and vice-versa. This is the case if the operation extensions define exact O-emulating functions. Since we require the detection to be immediate, values of type τ' must always be monitored. This implies then, that modules which define a dummy parameter D of type τ and which is to be associated to a carrier parameter of type τ' effectively redefine D to be of type τ', despite of the fact that this association is type-wise correct [theorem 5.1.1]. We have then:

Lemma 6.2.2 Let τ' be an extension of type τ such that τ' defines at least one detection extension not contained in τ, all modules defining dummy parameters of type τ to be associated to carrier parameters of type τ' must effectively retype the dummy parameters to τ'.

Observe that if τ' is not known statically, this may cause dynamic recompilation of the "receiving" module. Lemma 6.2.2 implies also that exception detection submodules

be explicitly defined as such, since, otherwise, the dummy parameter is not necessarily retyped.

Recall that change monitoring is in fact a "change completed" exception handling, where the "change completed" exception handler evaluates the detection condition and signals exception if the condition is true. Observe though, that the signal operation may yield control to another handler. This does not mean, however, that the signalling handler completed elaboration. That is, the handler activated due to the signal operation is in fact a "sub-handler" of the signalling handler. We have then:

Fact 6.2.3 Preventing exception handlers from generating exceptions restricts the power of expression of the language.

Now, if we allow an exception handler H to detect and signal exceptions of the same exception condition as that handled by H, infinite recursion loops may occur. Let us examine then whether such conditions could be forbidden at compile time.

Consider the case of the overflow exception condition. Several different operations in, again several different situations, may cause overflow to be detected. The actions to be performed when an overflow has been detected do not necessarily have to be equal for all operations and situations. For example, within a given expression it could be valid to use the largest real value, e.g. max real, as the result of a real addition which caused overflow. In another expression, or point within the same expression, a real addition overflow may be an error condition. Now, if the decision of which handler to chose is to be made without dynamically redefining handlers, it follows that the selec-

tion must be based on environment information. For example, by means of an associative table relating code points and handlers, the handler could be selected by simply searching for the entry which code point is equal to the detection point. Notice, though, that such a selection procedure is part of the exception handler even when implied by the language processor. Furthermore, all effective handling sub-modules are also part of this exception handler.

Defn. 6.2.4 An exception family F of an exception condition E, is a set of exception conditions $F=\{F_1,F_2,\ldots,F_n\}$, where F_i is detected when and only when E is detected and the detection condition of F_i is met.

From the discussion above it follows:

Fact 6.2.5 Preventing the definition of exception families restricts the power of expression of the languages.

The detection condition of a member F_i of an exception family F could be defined by the user and be textually placed within the family exception handler H of exception E. For example, H could determine what action is to be taken when a run-time error is detected by consulting an error code, e.g. ONCODE in PL/1, W28 in IPL V etc. It follows then, that it is undecidable to know, in the general case, which member F_i of the family has effectively been detected, since this decision may be elaboration dependent. Now, detection of a member F_j when a member F_i is being handled by a handler instance H_1 of H does not necessarily induce an infinite loop, since when F is finite and always $F_i \neq F_j$, eventually the recursive detection must end. Finally, if the handler instance H_k which handles the recursively detected exception F_j allows H_1 to recover from this exception, H_k may be invoked, since the elaboration of

H_1 will still be meaningful. It follows from this discussion:

Fact 6.2.6 Preventing an instance H_k of an exception handler H from detecting an exception E which will be handled by another instance H_1 of H, restricts the power of expression of the languages

It follows from fact 6.2.6 that the programmer must be made responsible for the avoidance of recursive exception handling. Thus, in order to avoid unexpected recursion, exception handlers should be explicitly declared as being recursive, allowing then the language processor to provide automatic testing of an illegal recursion attempt.

Detection of an exception is not necessarily linked to the detection of the cause of the exception. That is, there may be an undefined time interval between the instant when an action which eventually leads to the detection of an exception is performed, and the instant where this exception is actually detected. These cases are particularly common to run-time error conditions.

The determination of the cause which lead to a given exception depends on the particular program being elaborated. It follows then, that users themselves must determine the causes of exceptions, i.e. provide exception diagnose. Several methods have been designed to provide early detection and possible diagnose of errors. Among these are: performing several redundant operations and comparing the results[Elm1]; mechanically testing whether certain conditions (assertions) are true at given points[Flo1,Hoa2].

Errors are frequently diagnosed by means of postmortem information, e.g. dumps[Bay1]. This information fails to

show, however, the dynamic evolution of program sections. Among the tools which aid in describing the dynamic evolution of program sections we will find the "most recent history of changes" of a given textual name. This history keeps track of a usually bounded collection of values which the data spaces bound to a given textual name possessed. Notice that some systems implement the most recent history of changes. For example, the Honeywell 6050 automatically keeps the 16 latest values the instruction counter possessed[Hon1]. Audits[Con1] and traces also show the dynamic behaviour of a program. However, they have the side effect of generating quite large amounts of usually unnecessary information.

When programming in a high level language we do not want to diagnose errors at the machine level. Furthermore, the histories to be kept and the amount of detail with which they are to be kept may vary from program to program. It follows then, that history keeping should be programmable.

In figure 6.2.3 we show how the most recent history keeping type "history" could have been defined. Observe that in fact this type defines a specialized trace function, which performs history keeping actions whenever a store operation is performed. The history keeping storage device "save_area" is a circular buffer. Such a storage device defines a bounded set of storage cells, where the contents of the cell containing the oldest information is replaced whenever a store into the circular buffer is performed, when no empty cell is available. Values of type textual_name are strings of characters and correspond to user defined textual names where the access function parameters are replaced by their elaborated values. For

```
type history(integer size; type monitor_type, save_type)=
   extend monitor_type with
      begin extension;
         circular_buffer(size) of save_type save_area;
         emulate store(monitor_type textual_name target)=
            save_area.to_buffer(convert(value::target));
         outside_scope reader;
            save_type fetch function read=
                  save_area.from_buffer;
         end reader;
      end extension;
```

Figure 6.2.3 Definition of the most recent history keeping type "history".

example, if J=2, the textual_name typed value of J is 'J' and of A[J,J*4] is 'A[2,8]'. It should be clear that, given a textual name typed value T, the contents of the data space bound to T are retrievable. The function convert stands for a system defined function which selects and elaborates a system or user defined conversion function with exactly the same range and domain types as those abstracted from the text. It should be clear that convert succeeds iff exactly one conversion function can be selected.

In figure 6.2.4 we show two examples of how the type "history" of figure 6.2.3 could have been used. The first example should be self-evident. In the second example we keep track of the changes to elements of an array, keeping only one stack though. The type "detail" describes the information which should be kept by "history". Since we want to know which particular element of the array has been changed, "detail" must define the pair <"name","value">,

```
history(size::32; monitor_type::save_type::integer) I;
 - - - - -
type detail(type detail_type)=
   begin detail;
      struct(textual_name name; detail_type value) element;
      outside_scope conversion;
         conversion(from::textual_name detail_type;
                    to::detail; value:: val)=
            begin conv;
               element.name=val;
               element.value=value(val);
            end conv;
      end conversion;
   end detail;

history(size::100; monitor_type::real;
                   save_type::detail(real)) A[5,5];
```

Figure 6.2.4 Examples of the use of the type "history".

where "name" is the textual name of the changed array element, and "value" is the value assigned to this array element.

Exception handlers may require information, say *exception information*, which is defined at the instant an exception is detected. For example, a given overflow handler may require the statement number and the textual names of the operation and corresponding arguments defined at the detection point. It should be clear that each exception (i.e. detected instance of an exception condition) defines its own exception information. Furthermore, the exception information may be *volatile* in the sense that, if not consumed within a given time interval, the exception infor-

mation may change and, consequently, cause the exception handler elaboration to be meaningless. We will study these aspects in section 6.4.

It should be noted here that some of the required exception information may be quite unusual from the point of view of conventional programming languages. For example, the first overflow handler we have mentioned requires statement numbers and textual names of the operations and their arguments. We will not enter into further details with regard to the set of primitive types and language defined values which should be made available to the user.

The exception information requirements may change from handler to handler, even if each of these handlers handles exceptions of the same exception condition. Since different exception information requirements may require different compilation (elaboration) strategies, it follows that all exception handlers which might be invoked within a given program section must be known when this program section is being compiled. This implies then, that program sections which may cause an exception handler to be invoked be qualified with the exception information requirements of that handler. The same is true also for the detection information. We have then:

Fact 6.2.7 Exception descriptors must define both the detection and the exception information required to invoke the corresponding handler. Furthermore, the program sections which are to be serviced by a handler H, must be qualified by an exception descriptor which describes the information required by H.

Since both detection and exception information are defined within exception descriptors, it is natural then to define also the textual name of the exception handler which services the exception defined by the descriptor. Now, these textual names could define module typed variables, e.g. "stack of module". In doing so, we are able to assign an exception handler H, i.e. a module typed value, to this variable, where H is the most adequate handler relative to the program context where the corresponding exception could be detected. Notice that such a module typed variable could have been initialized when the descriptor instance is created.

It should be clear that the exception information is received by the exception handler by means of a dummy parameter list. Now, if for each dummy parameter there is a default parameter value, the corresponding carrier parameter list could be empty. Notice that this corresponds in fact to the implicit definition of the carrier parameter list. The default parameter values could be generated by user defined functions and thus impose virtually no restriction on the kind of values which could be part of the exception information. Finally, the exception information can be abstracted from the default parameter values, since they could be defined in terms of information available at the detection point.

Exception information is not always readily available, i.e. some computation may be required in order to produce this information. This may be the case also when the exception information is system defined. For example, textual names may have to be read from secondary storage in order to reduce the amount of usually unnecessary information in main storage. Notice that a similar approach has been

defined for the symbolic dump of the ALCOR compiler[Bay1]. The difficulties caused by the existence of these time intervals will be discussed in section 6.4.

```
when (real_overflow):
   real ovflw(integer stat#=statement_number;
              textual_name op=operation,
              names[degree(operation)]=arguments)=
      begin overflow;
         output ' ! Overflow ', stat#, op, names;
         return(fail deactivate real_overflow(stat#,op,name));
      end overflow;
```

Figure 6.2.5 Example of an exception descriptor.

In figure 6.2.5 we show an example of an exception descriptor. The textual name "ovflw" defines a module typed variable which is initialized to the module typed value contained within the begin end bracket pair. Observe that the handler "ovflw" is defined in the same way as a conventional procedure. However, the when prefix denotes that it is not to be activated by means of conventional procedure calls, but, rather, that it is to be activated when the respective exception is detected or signalled.

In section 6.3 we will discuss how exceptions are transmitted between modules. In particular we will examine the need and meaning of the constructs such as return and fail deactivate when used within exception handlers. We will anticipate the description of these constructs in order to be able to examine the information flow aspects when an exception handler passes an exception on to another handler, or when an exception is explicitly signalled.

The fail deactivate construct deactivates the exception handler containing this construct. Furthermore, the fail deactivate itself signals an exception, as it is the case of figure 6.2.5. Although not necessarily, there is then an exception handler H (possibly the same as the one which signalled the exception - see fact 6.2.6) which accepts and services this signalled exception. This exception handler H could generate a value which should be used instead of the value generated by the operation preceding the detection point. For example, in figure 6.2.5 this value could be the max real value. Thus, it is possible that a signal operation returns a value. In figure 6.2.5 this value is then returned to the preempted module by means of the return construct.

It should be clear that the exception handler H which services the signalled exception not necessarily returns control to the module signalling this exception, e.g. the "ovflw" handler in figure 6.2.5. It follows then, that the preempted module and the deactivated handler will eventually have to be terminated by means of a garbage collection, since the elaboration of the program may reach a point where reactivation of these deactivated modules is impossible. We may conclude then, that the module instance retention mechanism as defined by Johnston[Joh2] and Berry[Ber2,Ber3,Ber4] is the natural module instance termination mechanism, instead of the stack mechanism used in most of the current laguage implementations[Dij1].

Observe that, when explicitly signalling an exception, the exception information is usually not to be abstracted from the detection point, i.e. the signal operation. It follows then, that the signal operation must be capable of transmitting information by means of a carrier parameter

list. Notice that the existence of carrier parameters overrides the default parameter value initialization of the corresponding dummy parameters. Thus, the information provided by the signal operation can be effectively transmitted to another handler. Since the signal operation itself does not know the handler which will be activated, it follows that the dummy parameter names of the handler servicing the signalled exception must adapt to the carrier parameter names of the signal operation. In section 4.3 we will show that there is a mechanism linking exception descriptors to handlers, i.e. module typed values. Since this mechanism must know both the exception descriptor and also the corresponding handler, this mechanism is capable of defining a name equivalence [see section 5.3] which relates the parameter names of the carrier, i.e. signal parameter list, and the dummy parameter lists, i.e. exception handler parameter list.

Exception descriptors are information and are kept in data spaces of type exception. It follows then, that exception descriptors are identified by exception typed textual names. It is possible then to qualify an operation or a program section by such a textual name E and, thus, state the exception information requirements of this exception descriptor E. Such a qualified operation or program section will said to be E-qualified.

Different operations or section of a same module could be qualified by different exception descriptors based on the same exception. For example, a given expression could contain several different operators all of which detect underflow, for some of these operators the replacement of the result by 0.0 could be valid, whereas for other

operators such a replacement could induce numerical inconsistencies, e.g. division by zero.

It should be clear that exception descriptors which are not operation dependent, e.g. an external interrupt, should qualify entire modules rather than operations. On the other hand, exception descriptors which are operation dependent, e.g. overflow, actually qualify only those operations in a program section which could detect the corresponding exception, regardless of the qualification scope.

```
exception ov1=when(overflow) ...
exception ov2=when(overflow) ...

 Examples of use:

ov2:(E+F) ov1:/ (H ov2:+Z)
```

Figure 6.2.6 Examples of E-qualified sections and operations.

In figure 6.2.6 we show some examples of E-qualified sections and operations. The exception descriptor "ov2" qualifies both the balanced expression "E+F" and the operator "+" in the subexpression "H+Z". Finally, the exception descriptor "ov1" qualifies just the division operator.

Not always do exception conditions apply to global operators such as the "/" operator in figure 6.2.6. For example, a type descriptor might define several operators which are non-standard and which are not known to the exterior of this type descriptor. It follows then, that the exception descriptor as well as the handler should be part of such a type descriptor.

```
extend array with begin extension;
   queue(struct(integer wrong_index, wrong_val)) wrong_list;
   boolean access_fail:=fail;
   exception sr = when(subscript_range):
      integer function wrong_found(
                        integer wi=subscript_index,
                                wv=argument)=
         begin wrong_found;
            wrong_list.to_queue:=build(wi,wv);
            access_fail:=true;
            return lower_bound[wi];
         end wrong_found;
   emulate test_subscript = sr: perform test_subscript;
   emulate access = begin access;
      if access_fail
         then /* output messages, "wrong_list" etc. */
              wrong_list.empty;
              access_fail:=false;
              fail deactivate access_error;
         else perform access;
      fi;
   end access;
end extension;
```

Figure 6.2.7 Example of an exception descriptor embedded in a type descriptor.

In figure 6.2.7 we show how an exception descriptor, i.e. "sr", could be defined internally to a type descriptor, i.e. extension of "array". The subscript range checking of the subscripts of an array could be performed as the individual subscript expressions are evaluated. In many cases though, e.g. when side effects exist, all

subscript expressions must be elaborated previous to the preemption of the module containing the offending access to the array. It follows then, that the subscript_range exception handler must allow a recovery, however, it must also effectively prevent the array from being accessed. In figure 6.2.7 this is achieved by returning a valid value of the subscript and by setting the "access_fail" flag to true, consequently preventing the access to the array from being performed. The queue "wrong_list" is empty and the flag "access_fail" is false whenever control is outside the extended "array" type descriptor. Thus, two different accesses do not interfere with each other.

Since, in principle, all operations are defined within some type descriptor, exception handling facilities can be hand taylored for each individual case by means of some type descriptor extension as shown in figure 6.2.7, as long as such an extension is permitted. For example, run-time detected type mismatch could be dealt with by extending all type descriptors which define operations binding access functions to data spaces, e.g. the parameter list association operation. It follows from this discussion that the language should describe all data spaces and operators which could be replaced and/or accessed by the user in order to get more precise information about a given exception. For example, in figure 6.2.7 we assume the existence of the "subscript_index" counter which defines the index of the subscript expression being evaluated when the subscript_range exception is detected. As said before, we will not enter into details with respect to the set of laguage defined values which should be made available.

6.3 Exception Handler and Program Interaction.

In this section we will examine how exception handlers interact with the program being elaborated. There are two cases to be considered:

a- how do the exception handlers interact with the detection point containing module, say the preempted module;

b- how do the exception handlers interact with other modules in order to service the detected exception.

Let us examine first the interaction of the exception handler H with the preempted program section S. There are three cases of interaction to consider:

i- H deactivates and elaboration of the preempted module resumes at the detection point, e.g. immediately after the operation which signalled the exception. We say in this case that the preempted module has recovered from the detected exception, and that H is a recovery handler.

ii- H deactivates and transfers control to some predefined point within the preempted module.

iii- H deactivates or terminates passing control to another module which is not the preempted module.

It should be clear that case (i) corresponds to a disguised subroutine call. In some cases a return value may be expected from the exception handler. For example, when handling underflow, it is frequently satisfactory to replace the result of the underflow signalling operation by 0.0. The notation introduced in section 6.2 treats exception handlers as if they were conventional modules, e.g.

subroutines, coroutines or functions. We will use then the same constructs as those developed for conventional modules in order to return control to the detection point when the handler finishes elaboration.

It should be noted that the detection point may have to be defined explicitly in some cases, c.f. section 6.2. For example, the "operation" which causes the detection could be a macro expansion and, thus, the actual detection point could lie within this expansion. However, when returning to the detection point, the elaboration should resume at the code point following the macro expansion. This case is in fact a subcase of the "transfer to a predefined point" case discussed in the sequel.

```
exception uf=
   when (underflow):
      real uflw(integer stat#=statement number)=
         begin underflow;
            output ' ! underflow at ', stat#;
            return(0.0);
         end underflow;

 Example of use:

... a+ uf:((x+y)/z) *q ...
```

Figure 6.3.1 Example of a value returning exception handler.

In figure 6.3.1 we show an example of a value returning exception handler. This handler produces a message and returns the value 0.0 whenever an underflow is detected within the "uf" qualified section "(x+y)/z".

Consider now the case of elaboration resumption at a predefined point (case (ii)). The exception handler H is

deactivated (or terminated) when it transfers control to a predefined point, where such a predefined point should not be viewed as a subroutine or coroutine entry. That is, a transfer of control to a predefined point does not record the origin of the transfer of control in the general case.

Defn. 6.3.1 A retrial point is a code point to which an exception handler H may transfer control when deactivating, where this retrial point is not necessarily equal to the detection point.

Although the transfer of control to a retrial point does not mean that some program sections are necessarily reelaborated, it has the power to do so. This justifies then our nomenclature.

It should be noted that the transfer of control to a retrial point corresponds to a goto. Such a goto is particularly bad since it is completely hidden within the text. Let us examine then how we could make the flow of control paths more explicit.

We will say that an exception handler H induces a retrial loop, if H transfers control to a retrial point r, such that the elaboration starting at r may trace the same detection point which caused the activation of H.

It is a common practice that input operations which cause input error exceptions are retried, a definite error being reported only when more than a predefined number of retrials have been performed. It follows then:

Fact 6.3.2 Preventing the existence of retrial loops restricts the power of expression of the language.

Notice that retrial loops could be induced indirectly by means of a while loop driving boolean variable set by the handler. It follows then, that it is undecidable in the general case whether a given exception handler is part of a retrial loop, even if this exception handler is a recovery handler.

Suppose now that we would restrict each detection point to detect at most one exception within the time interval during which the corresponding module instance exists. This solution is non satisfactory, since retrial loops could iterate for several times until the exception is eventually "cured". For example, in the input error handling loop mentioned above, we iterate a bounded number of times until either no input error is detected or the preassigned maximum number of iterations is exausted. Furthermore, the number of iterations of a retrial loop may be program dependent. For example, in order to achieve a highly available and reliable system, Elmendorf suggests that intermittently faulty program sections be reelaborated a given number of times, where reliability is a function of this number[Elm1]. We may conclude then:

Fact 6.3.3 Imposing a system defined restriction on the number of detections of an exception c at a detection point p, restricts the power of expression of the language.

This points then into the direction that the exception handler itself should be made responsible for the termination of any eventual retrial loop. The loop termination could be assured implicitly if it is known that the exception source has either been removed, e.g. corrected, or the preempted program section will not be reelaborated, e.g.

assured termination of the module instance. The retrial loop termination may also be assured explicitly by means of a retrial loop control generator function, where the body of this generator function is the exception handler. Such a generator function is necessarily a coroutine, since deactivation of the handler implies also a deactivation of the loop control generator funtion, however the internal state of the generator function must be kept intact until the next activation of the handler, since, otherwise, the retrial loop cannot effectively be controlled.

The retrial loop controlling generator function must be initialized when, or before, the first exception is detected. This initialization could be dynamically requested. For example, we must reinitialize the input error counter whenever a new input record is requested. Now, a given retrial loop controlling generator function could control several different detection points, since these detection points could occur in several different E-qualified sections, all of which are qualified by the same exception descriptor E. It follows that we must provide a construct which allows to explicitly initialize a given retrial loop controlling generator function, i.e. initialize. Such an initialization request is not necessarily expensive, since the generator function could define an "initialization flag" which forces the generator function to be initialized iff this flag is off when an exception is effectively detected.

In figure 6.3.2 we define the retrial loop inducing exception handler "reread". The retrial loop may be performed at most 20 times, after which a "read_error" is signalled for the file parameter "file" of the exception descriptor "read_again". Notice that the "file.input_error"

```
exception read_again(file file)=begin read_again;
   retry for I:=1 until 20 do;
      when (file.input_error):
         reread=begin reread;
            /* perform recovery actions on "file"*/
            retry;
         end reread;
      od;
      fail deactivate(file.read_error);
      return;
end read_again;
```

Example of use:

```
initialize rd_ag=read_again(some_file);
retry(rd_ag): read(some_file) into area;
```

6.3.2 Example of an explicitly controlled retrial loop.

exception handler is the body of the generator function, i.e. for loop, and that it defines the condition of the exception descriptor. It should be clear that the when construct defines a (coroutine) activation entry rather than a creation entry. Syntactically this can be abstracted from the existence of the retry verb at the head of the handler. Finally, the only replaceable portion of the exception handler is the procedure "reread". That is, "reread" is the textual name of the exception handler defined by the descriptor "read_again". Observe that the verb initialize dynamically declares a coroutine instance as required by section 4.5.

Due to the parameter "file" the exception descriptor "read_again" is bound to a given file. It follows then, that the exception descriptor "rd_ag" implicitly declared

in the initialize statement handles only input_errors which are detected when reading records (blocks) from the file "some_file". Now if the initialize statement is provided for every read statement relative to "some_file" there will effectively be 20 record reading retrials before a "read_error" is signalled by the handler relative to the file "some_file".

Due to the disjointness of the initialize construct and of the retry qualifications, a single initialization for several different operations is possible. For example, when handling underflow it may be valid to replace the result of the underflow signalling operation by 0.0. However, when doing so, numerical precision may be lost and, thus, there is a bound on the number of such replacements when a certain precision must be assured.

```
exception ufb=begin bounded_underflow;
   when(underflow):
      real co_function uflw(integer stat#=statement_number)=
         begin underflow;
            integer I initial 0;
            while I<10 do;
               I:=I+1;
               output ' ! ', I, ' underflow at ', stat#;
               deactivate(0.0);
            od;
            fail return underflow(stat#);
         end underflow;
end bounded_underflow;
```

Figure 6.3.3 Example of a record keeping exception handler.

In figure 6.3.3 we show how the underflow handler mentioned before could have been defined. In this example the exception handler itself keeps the record rather than the exception descriptor as in figure 6.3.2. This example shows also how exception handlers could be declared as coroutines. Recall that the statement "deactivate(0.0);" is a coroutine deactivation [see section 4.5]. Observe also that the exception handler "uflw" initializes itself when being created, thus not requiring an explicit instance declaration.

It should be clear that a retrial point is in fact the value contained in a retrial point typed data space. Furthermore, these values are closely related to label typed values. It follows then, that retrial point defining constructs assign values to retrial point typed data spaces, where such a value is the label of the program point where the retrial point definition occurs.

Let α be a retrial point typed data space. The contents of α could be defined dynamically as it is usually the case when retrial loops are established. They could also be defined statically as it is usually the case when exceptions terminate sub-modules. It should be clear that the positioning of retrial points is program dependent, since this positioning must be such that a meaningful elaboration results. It follows then, that there must be a construct by means of which a retrial point typed value is defined. Furthermore, the property of being statically or dynamically defined is also program dependent, consequently the construct mentioned before must distinguish between static (retry) and dynamic (dyn retry) retrial point definitions.

A given exception descriptor could make use of several retrial points. It follows then, that retrial points should be named. This name could be the empty string as in figure 6.3.2. Suppose now that we restrict handlers to use only retrial point typed names defined within the exception descriptor of the activated exception handler. This does not restrict the power of expression of the programming language, since the computation which follows a transfer of control to a retrial point must be well understood by the exception handler in order to be meaningful. It follows from this restriction that we are able to determine exactly the program conditions which are satisfied at a retrial point definition point as required by some program proving methods[Hoa2].

```
exception reshape=begin reshape;
   retrial_point r;
   retry when(subscript_range):
      rebuild(...)=begin rebuild;
         /* reorganize the faulty array */
         retry r;
      end rebuild;
end reshape;

Example of use:

integer A[1,1], B[1,1,1];
reshape: A[f(x) dyn_retry r(reshape),
                dyn_retry r(reshape) g(y)];

B[dyn_retry r(reshape) i reshape: , j, k reshape: ];
```

6.3.4 Example of retrial point definitions.

In figure 6.3.4 we show an example of a dynamically defined retrial point. The purpose of the exception handler "rebuild" is to redefine arrays whenever subscripts are out of bounds. Thus these arrays become adaptive to a given elaboration's needs, e.g. ALGOL68 flex array bounds. The exception descriptor "reshape" defines a retrial point "r". The construct "dyn retry r(reshape)" causes the value contained in the data space α bound to "r" to be set to refer to the point in code where this construct occurs. Notice that the subscript range test is an implicit operation and, thus, is textually non-existent.

In the example using the array "A", each dimension is retried individually. Notice that "f(x)" will not be reelaborated, however "g(y)" will be reelaborated whenever an exception is detected in the corresponding program section.

In the example using the array "B" the subscript range exceptions will be handled by "rebuild" only if the first or the last dimensions are out of bounds. In both cases though, the whole subscript list will be reelaborated. It should be noticed that the "reshape" exception descriptor does not have to be initialized, since there is no loop control generator function. Although syntactically these examples could be termed "ugly", we will not attempt to define a more elegant syntax in this dissertation.

The last case of interaction between the exception handler H and the preempted module M, is when the handler H deactivates without returning control to the preempted module M (case(iii)). Observe that such a deactivation could be generated artificially by defining a program section S within M, where S contains a retrial point defi-

nition and an M deactivating statement. However, such an artifice makes the reading of a program more difficult since a given action is now spread over the program's text. We define then the failure deactivation constructs fail deactivate and fail return, which have the power of deactivating the exception handler without returning control to the preempted module M, in the rest being similar to the success deactivation constructs deactivate and return. It should be clear that the preempted module M is deactivated when passing control to the exception handler H.

Usually a failure deactivation is in itself an exception and, thus, signals another exception. In some cases though, a failure return may stand for a "normal" termination of the module M. For example, the detection of "end of file" could represent the termination of a compiler coroutine which scans successive syntactic units in the input stream[Con2]. We will adopt then the convention that a failure deactivation corresponds also to a signal operation, but only if a condition name is present in the failure deactivation construct. Recall that the condition name could be followed by a carrier parameter list as described in section 6.2.

In the case a failure return corresponds to a normal termination of the preempted module M, the deactivation must occur through an entry point of M as described in section 4.5. Of course, this implies that the exception handler (descriptor) must know this entry point, or then transfer control to a retrial point which eventually causes M to be deactivated. Thus, eventual naming conflicts could be resolved by means of a name equivalence (see section 5.3) when assigning a module typed value to the handler typed variable.

In figure 6.3.2 we show an example of an exception signalling failure deactivation. If the "read_error" handler performs a recovery, elaboration resumes following the deactivation point within the handler "reread" and, thus, resumes at the detection point of the preempted module.

Even when the preempted module M is a subroutine, it does not have to be terminated when a fail deactivation of M occurs. This follows from the fact that M could define several access functions by means of which a more detailed information about the exception could be obtained. It follows then, that subroutine instances should not necessarily be terminated when a failure deactivation occurs.

Let $U=\{u_1,u_2,\ldots,u_n\}$ be a set of exceptions which could be detected within a module M but for which M does not define exception descriptors. This set of exceptions establishes an exception family, say the unknown exception family. It follows then, that M could define an unknown exception handler H†. Notice that the unknown exception handler must be defined within the dispatcher module of the operating system, in order to catch all non serviceable user defined exceptions, since, otherwise disastrous termination of the operating system could occur.

Defn. 6.3.4 An exception u is unknown to a module M, if u can be detected within M and signalled to the exterior of M.

Observe that an exception u can be unknown even if M defines a handler H for this exception. This follows from the fact that the exception u may be signalled to the exterior of M, despite of the fact that M defines the

---

† Notice that unknown is the name of the exception (handler), thus the unknown exception (handler) is in fact known!

handler H. Thus we may regard this handler H as being a default handler. As we shall see later in this section, it may be necessary to override local or, better, default exception handler definitions.

It should be clear that an external handler H servicing a signalled exception u must have been created, at least partially, when the exception u is signalled by the module M. Of course, there could be a generalized exception handler $\hat{H}$ which purpose it is to receive all exception signals and direct control to the appropriate handler H, where H could be a subroutine of $\hat{H}$.

Let u be an exception which is unknown to the set of modules $M=\{m_1,m_2,..,m_k\}$. Let $H=\{h_1,h_2,...,h_l\}$ be the set of exception handlers capable of servicing the exception u. We must decide now which (one or more) of these exception handlers h_j should be activated when a module m_i signals u.

The exception u could represent a non-recoverable error. Thus, continuing with the elaboration of the preempted module m_i would be meaningless. Now there could be an exception handler h_j which attempts recovery, since not necessarily is u non recoverable for all modules in the set M. For example, overflow could be recovered in some cases, e.g. by replacing the result of the operation by max real, whereas in other cases the result may become meaningless if this recovery is attempted. It follows then, that not all members of the set H are eligible to be activated when a module m_i detects u.

Defn. 6.3.6 An exception handler H which allows a module M to recover from an unknown exception u is logically correct, iff the actions performed by H maintain the correctness of the elaboration of M.

It should be clear that any recovery must be logically correct. Notice though, that logically correct recoveries may decrease the quality (precision) of the results produced by M. For example, when substituting the result of an overflow detecting operation by max_real, the numerical accuracy may be decreased but not necessarily so that the numerical result be incorrect. Thus we must view the term "correctness" in a broader scope as meaning "satisfying the admissible tolerance".

In order to decide whether a given recovery handler H is logically correct with respect to a given module M, H must perfectly understand M, since the correctness (precision) of M must be preserved by H. Since the correctness of M is undecidable in the general case, it follows:

Lemma 6.3.6 It is undecidable in the general case whether a recovery handler H is logically correct with respect to a module M.

In section 6.2 we have introduced the concept of E-qualified sections or operations, where E is an exception descriptor defining the detection information and condition, the exception information, and the textual name of the exception handler, where this name is possibly initialized to a default parameter value, i.e. module. We have mentioned also in section 6.2 that there could be several different exception descriptors, all of which are defined within the same module M and describe basically the same condition. However, each of these exception descriptors E_i implicitly conveys more precise information with regard to a given condition detected within an E_i-qualified section of M, than that conveyed solely by the exception signal. For example, we could define two overflow descriptors ov_1

and ov_2, where ov_1 qualifies all those operations where a recovery would be valid in principle, and ov_2 qualifies all those program sections where such a recovery is meaningless. Formalizing we have:

Fact 6.3.7 The exception signal s must define both the exception condition and the exception descriptor E which qualifies the program section within which s has been detected.

It follows then, that the interface information of a module M must define all those exception descriptors for which M allows an explicit external handler, i.e. a non unknown handler, to be defined. Since exception descriptors could be variables of type exception, where these variables are initialized by means of default parameter values, it follows that we have also a mechanism by means of which an exception descriptor could be substituted and, thus, exception information requirements could be changed whenever necessary.

It should be clear that, when substituting an exception descriptor E_1 by another descriptor E_2, the exception condition c of E_1 must be kept invariant. It follows then, that the type exception is in fact a composite type defining <exception descriptor, condition value>. In section 3.4 we have discussed the problem of type equality, thus we need no further explanation with respect to assuring the invariance of the exception condition, when replacing exception descriptors.

Theorem 6.3.8 A recovery handler H of a module M detecting an unknown exception u, may be activated in the general case only if H is explicitly related to an exception descriptor E defined by M.

Proof. By lemma 6.3.6, if H is not explicitly related to M, we are unable to decide, in the general case, whether H could be logically correct with respect to M. By fact 6.3.7 the logical correctness depends on the exception descriptor E which qualifies the program section or operation where the exception has been detected.

We cannot assure that H is logically correct with respect to E whenever H and E are explicitly related, since either or both the handler H and the relation of H to E could be in error. Apart from these programming errors though, we could assure the logical correctness of H with respect to E. It follows then, that, under the assumption of correct programs, theorem 6.3.8 could state a necessary and sufficient condition.

Defn. 6.3.9 Let $R=\{r_1,r_2,\ldots,r_n\}$ be a set of exception handlers which allow a module M detecting an unknown exception u to recover from u in a logically correct fashion. An elaboration of M is said to be maximal iff, whenever M detects u, one or more of the handlers r_i in R is activated.

As mentioned above, the set R is not necessarily equal to the set H of exception handlers capable of servicing u. Maximizing the elaborations of modules is a desirable feature, since meaningful elaborations will terminate successfully.

It follows from theorem 6.3.8 that an exception u signalled by a module M must define a pair <u,E> where u identifies the exception and E the exception descriptor qualifying the detection point. However, if there is no handler capable of servicing <u,E>, an unknown exception

handler must be defined for <u,E>. This follows from the fact that we want to maximize the elaboration of the program, consequently we do not wish that an exception, for which there is no specific handler defined, is able to halt the elaboration of the program. Formalizing we have:

Theorem 6.3.10 Let M be a module containing an exception descriptor E describing an exception condition u. The elaboration of a program containing M is maximized when M signals u detected within an E-qualified section only if:

i- there is a <u,E> exception handler H capable of servicing <u,E>; or if such a handler H does not exist:

ii- there is an unknown exception handler U which handles either the <u,unknown> exception or the unknown exception.

Theorem 6.3.10 defines the address of the "receiver" exception handler when exceptions are signalled. Theorem 6.3.10 states also that modules could always define an unknown exception handler, e.g. system defined handler, since this handler will only be activated if no handler servicing the exception u has been defined. Observe that the program sections labelled SYSERR in ALTRAN are in fact such unknown exception handlers.

In order to be able to establish an information interchange linkage between the preempted module and the servicing handler, the exception descriptor identification may prove to be insufficient information, since there could be several instances of a same module, some of which may simultaneously be in the preempted state. Thus, in order to access specific information, e.g. by means of special

access functions, the specific module instance must be known. We have then:

Fact 6.3.11 In order to establish an information interchange linkage between the handler and the preempted module instance, it must be possible to abstract the identification of the preempted module instance from the exception signal.

Notice that such an information interchange linkage may be required also in order to produce error diagnostics, or to decide about further action in the case of an exception which signals a successful conclusion of the elaboration of a module.

We have now to examine the case when two or more exception handlers are eligible to service a given exception u, i.e. $\langle u,E \rangle$. Let P, Q and R be three modules such that P activates Q and Q activates R. Suppose now that both P and Q define eligible exception handlers H_P and H_Q for the exception u signalled by R. Furthermore, we may safely assume that any recovery of R from u induced by either H_P or H_Q is logically correct. Now, if H_P is a recovery handler and H_Q is a failure handler, we must chose H_P in order to maximize the elaboration of P, Q and R. Formalizing and by induction we have then:

Lemma 6.3.12 Let $H=\{h_1,h_2,\ldots,h_n\}$ be a set of exception handlers capable of servicing an exception u signalled by a module M. Imposing a system defined order of preference for the choice of a handler h_i in H when u is signalled by M, may prevent the elaboration of M from being maximal.

Lemma 6.3.12 tells us that the dynamic scope of exception handlers should not be suppressed by system action. Thus, there may be more than one eligible handlers at given time instants. Some of these handlers may be unsatisfactory though, since the quality degradation could be excessive for a given application. Since only the programmer can decide the admissible quality degradation, it follows:

Theorem 6.3.13 Let M be a module containing an exception descriptor E describing the exception condition u. The elaboration of M can be maximized, in the general case, only if E can be explicitly linked to one or more exception handlers all of which will be activated when u is detected within an E-qualified section of M.

Let $H=\{h_1,h_2,\ldots,h_m\}$ be the set of exception handlers which are capable of servicing an exception u, and let $E=\{e_1,e_2,\ldots,e_n\}$ be a set of exception descriptors which qualify program sections signalling the exception u. The linkage definition linking handlers to descriptors, defines a bipartite graph G, where the vertex sets are the sets H and E, and the edge set is the set of all linkages which could be established between any two h_i and e_j for all possible elaborations of the program. The pair $\langle H',E'\rangle$, where H' is the handler set and E' is the descriptor set of a maximal connected component G' of G, will be called linkage set. Such a linkage set $\langle H,E\rangle$ defines a subprogram containing H and E, and is itself defined by a given linkage definition L. Suppose now that the linkage definition is not global to a given linkage set $\langle H,E\rangle$. There are then subprograms P' and P'' of the program P, which define linkage sets $\langle H',E'\rangle$ and $\langle H'',E''\rangle$ respectively. These linkage sets are not necessarily disjoint, and may

correspond to the independent linkage definitions L′ and L″. Thus a descriptor e_j could be linked to a recovery handler h_i by L′ and to a failure handler h_i by L″, both of which must be activated and which actions contradict each other. Formalizing we have:

Theorem 6.3.14 A linkage definition L linking a handler h_i to a descriptor e_j must be global to the linkage set <H,E> defined within the whole program and where h_i is in H and e_j is in E.

Theorem 6.3.14 tells us that there could be several independent linkage definitions, however each one being global to a given linkage set, where such a linkage set is maximal within the program. Since the linkage definition is confined to a linkage set, incorrect linkings of exception descriptors and handlers are local to a linkage set, thus preventing inconsistencies due to the existence of unknown or unforeseen exception handlers. Notice that theorem 6.3.14 is not satisfied by PL/1, since declarations of handlers (on-units) are local to modules and never global to a group of modules.

Observe that the linkage set does not necessarily restrain itself to the dynamic hierarchy of modules. That is, if a module R is a subroutine of Q which in turn is a subroutine of P, it is perfectly valid to have P and R in one linkage set and Q in another with respect to the same exception u. Furthermore, due to the direct binding of handler and exception descriptor instance, we are safe from difficulties arising from parallel or quasi-parallel elaboration with respect to handler definition.

Observe that the linkage definition performs in fact a module typed assignment, where the handler, i.e. a module

typed value, is assigned to the handler's textual name defined within the exception descriptor. Since handlers are values, it follows that quite complicated storage mechanisms, e.g. "stack of module", could be defined in order to store such values. These mechanisms have been studied in previous chapters.

6.4 Time Dependent Aspects of Exception Handling.

In this section we will study the problems which originate from the existence of time constraints on exception handling. In section 5.1 we have shown the basic principles which govern shared data space access synchronization. Thus we need not study in this section the timing aspects relative to the sharing of exception information. We will study then the timing problems which arise due to the occurrence of consecutive signals and due to the existence of a non-null time interval between detection and exception servicing.

Let H be an exception handler servicing an exception c_1 at the instant t. Suppose now that at this same instant t, the exception c_2 is detected and control is to be passed to H again. Now, if H is not reentrant, the two activations of H may interfere and, thus, produce meaningless results. We have then:

Fact 6.4.1 A module H may be activated due to an exception signal only if H is inactive or H is reentrant.

In sections 6.2 and 6.3 we have shown that programmers have the freedom to chose whether an exception handler is a coroutine or a subroutine, and whether the number of simultaneously existing instances of a handler is restricted. It

is thus the programmer's responsibility to decide whether the recurring detection of exceptions also causes a recurring creation of handler instances.

In most cases reentrancy is achieved by means of a module prologue which creates and initializes the necessary data spaces. This module prologue is non reentrant in most cases, consequently, by fact 6.4.1, the recurring activation of H must be postponed. Thus, once a handler H has been activated, recurring activation of H must be prevented, say masked, until an instant t' where recurring activations will not interfere with other activations of H still in progress. It follows then, that the language must provide a construct, e.g. unmask(<handler>), by means of which the handler is placed in a "reactivatable" state. If this construct were not available, each handler would be capable of handling at most one exception signal, since the activation of a handler also masks this handler.

The time interval during which an exception handler H is masked will be called masked interval. Each exception signal in a recurring sequence of exception signals could be linked to a different exception handler. It follows then, that not necessarily will an exception signal c_2 have to be delayed due to another signal c_1 of the same exception condition c currently being serviced by a masked exception handler. Notice tough, that linking an exception handler to a descriptor takes time. Furthermore, during this time the exception cannot be handled, since the linkage, i.e. the assignment, may be incomplete and, thus the handler activation may be misleading. It follows then, that it is not the module instance which is masked, but, instead, it is the textual name within the descriptor which stands for this instance which is masked.

Suppose now that an exception c is detected and would require a masked exception handler H to be activated. Since H is masked, the transfer of control may not occur. Furthermore, the exception signal may not be dropped, since, otherwise, valuable information may be lost. It follows then, that the exception signal c must be stored until H is unmasked. Since the detection could recur and/or several different exceptions could be serviced by a same exception handler H, it follows:

Lemma 6.4.2 Let H be the textual name of a handler defined by an exception descriptor E. H must provide a storage mechanism which stores all exception signals which are to be serviced by a module bound to H and which have been detected during a masked interval of H.

Since the storing of exception signals takes time, we are faced again with the same problem as the one we are attempting to solve. That is, for each exception signal a receiving data space must be immediately available, or the storage mechanism must be masked during the process of storing this exception signal. Even when hardware performed, the storing of exception signals will take time. Thus there is a critical frequency which defines the minimum time interval, say critical interval, between recurring detections of a same exception. It should be clear that the storing of each exception signal must take less time than the critical interval, otherwise uncontrollable interference between two different exception signals may occur. We will not enter into further details with respect to the critical frequency though.

Let M be a module which builds the image of an object on a display terminal. This module M could be activated by a generator function G, which object it is to change the position parameters of the object being displayed. The user has then the impression of a continuous object movement on the screen. Now, the generator function could be such that it can only be terminated by means of an external interrupt[Els1,Zel1]. This generator function termination condition has then the effect, that the user is enabled to decide which position of the object is best suited for his needs, as it is being generated on the screen. Now, the external interrupt could occur when M is being elaborated. If M were to be terminated immediately, an incomplete image may be placed on the screen. Thus M must always run to completion before the signal can be handled. We have then:

Fact 6.4.3 Enforcing exceptions to be serviced immediately when they are detected restricts the power of expression of the language.

Of course, the interrupt must be acknowledged and the exception signal must be buffered (stored) until the program elaboration reaches a synchronization point where the exception signal can be serviced. It follows from fact 6.4.3 that the language must provide a construct, e.g. mask(<handler>), by means of which a handler can be masked even if no exception has been detected.

It should be noted that the mask and unmask constructs are closely related to the data presence switch operations introduced in section 5.1. Thus mask and unmask can be viewed as operators which determine whether an information (module) is present (unmasked) or absent (masked).

Let c be an exception which is signalled whenever a module M consumes more resources than some predefined amount of these resources. For example, for the purpose of error diagnose we may wish to signal the exception c whenever the module M requires more than 90% of the total available dynamic storage. It should be clear that the elaboration of M is not affected by whether or not the exception c is actually serviced. Thus, even when the handler which would be activated in order to service c is currently unknown, the elaboration of M could proceed. It follows then:

Fact 6.4.4 Enforcing that a module M detecting an exception c may only proceed with its elaboration after c has been serviced, restricts the power of expression of the language.

Usually though, exceptions are serviced immediately. Furthermore, the decision of whether an exception must be serviced previous to the elaboration resumption of the pre-empted module is program dependent. For example, the result of an underflow signalling operation could automatically be replaced by 0.0 and, thus, it would be irrelevant in many cases whether underflow exceptions are effectively serviced. It follows then, that the language should provide a construct, e.g. fork signal(<exception>), by means of which the non-necessity of immediate exception servicing could be explicitly stated. It should be clear that forked exception signals start a quasi-parallel or parallel elaboration. Thus, there must be a mechanism by means of which the exception information can be saved whenever an attempt to change or destroy this information is made. This mechanism can be based on the data presence and data consumed

switches described in section 5.1, and, thus, needs no further examination here.

Since leaving a forked exception in a non serviced state, say pending, does not affect the elaboration of the preempted module, it follows that the nonexistence of a handler capable of servicing a forked exception at the detection instant is not necessarily an error condition. We may conclude then, that the assignment of a handler to the handler typed textual name of the descriptor of a forked exception might occur well after the exception has been signalled. Thus, forked exceptions must be buffered, even when no exception handler has been linked to such an exception. It follows from this discussion that whenever a handler is linked to an exception descriptor, it must be verified whether there are pending signals of the exception condition defined by this descriptor.

Let M be a module which detects an exception c which is left pending. Since the elaboration of M proceeds regardless of whether c is serviced or not, the handler H servicing c may not cause M to resume elaboration, since, otherwise, a terminated module instance could be activated. It follows from this, that we must be able to abstract from the exception signal whether it is a forked signal or not.

Let c be an exception detected by an exception handler H at an instant t where H is masked. If c may not be left pending and must be serviced by H, we have a deadlock situation. Not always are such deadlock situations as clear as in this example though. For example, the attempt to activate H could occur indirectly from within another module, e.g. exception handler. We will not study these

deadlock situations in this dissertation though[Hol1,Hol2, Hol3,Hab1].

The exception information may be volatile. That is, if the information is not consumed within a given amount of time, the information may be changed or destroyed. For instance, when reading information from a synchronous device, e.g. card, disk, tape etc., the information may be lost if the service time is too large, e.g. overrun exception on the IBM/360[IBM1]. We will say that an information I is real-time dependent, if there is a constraint on the length of the time interval $T=[t_1,t_2]$, starting at the instant t_1 where the information I has been produced, and ending at the instant t_2 when the information I has been consumed. The time interval T will be called real-time interval, and the interval T' which defines the maximum tolerable length of the real-time interval will be called real-time limit. If the real-time interval exceeds the real-time limit, we will say that the information, i.e. data space, has been overrun. Finally the instant t_2 will be called consumption instant.

The consumption instant is program dependent since it is not necessarily true that the first read access to a volatile information also consumes this information. It follows then, that volatile information must be explicitly flagged consumed as described in section 5.1.

Volatile information is the result of the inability to delay operations which change data spaces containing non-consumed information. It should be clear that this is hardly ever the case when no physical real-time constraints, e.g. process control constraints, are imposed on the elaboration. It should also be clear that the real-time limit

is not necessarily well defined. For example, when a channel can be attached to input/output units with different transmission rates, for each of these units the real time limit may change, although these units look alike to the channel. Furthermore, the real-time interval is not always well defined, since delays may occur due to unforseen interrupts. We have then:

Lemma 6.4.5 It is undecidable in the general case whether the real-time interval is smaller than the real-time limit for all possible interpretations of the program.

It follows from lemma 6.4.5 that there may be cases where the real-time interval exceeds the real-time limit. When this is the case, computations based on the corresponding volatile information are usually meaningless. It follows then, that a module making use of such a volatile information, e.g. an exception handler, must be preempted by an overrun signal.

It is a common practice to assign priorities to exception signals, where usually the priority is higher the smaller the expected real-time limit is. In doing so, exception handlers which are more critical with respect to time will not be preempted due to other less critical exceptions having been detected during the elaboration of these handlers.

In our generalized view of programs, it is perfectly valid for an exception handler to request part or all of the service to be done by other modules which may be elaborated in a parallel or quasi-parallel fashion. We will say that a module M is overrun dependent on a data space α containing volatile information, if the correctness

criteria of M may be affected by α being overrun, e.g. when M accesses α. It follows then, that the decision of whether a module is overrun dependent on a data space α is program dependent, since only the programmer is able to decide whether the correctness of a module may be affected by a data space being overrun. It follows from this discussion that modules must explicitly defined as being overrun independent.

A module which is not overrun dependent on a data space α cannot possibly be affected by an overrun on α, and, thus, may continue being elaborated. Furthermore, the elaboration of a module which is overrun dependent on α must be halted since the results will be meaningless in the general case. This follows from the fact that, usually, the correctness criteria of overrun dependent modules do no longer hold when α is overrun, since α may have been changed or destroyed when being overrun. We have then:

Theorem 6.4.6 Let α be a data space containing volatile information. If α is overrun, those and only those modules which are overrun dependent on α must be preempted.

It follows from theorem 6.4.6 that the overrun signal must have a higher priority then all overrun dependent modules which could potentially be active. Suppose it were not so. Then the overrun dependent module M could activate a module M′ which has a higher priority than the overrun signal and which performs essentially the same computations as M. Thus, M′ would continue being elaborated despite of the fact that M′ is producing meaningless results when an overrun has been detected. We have then:

Lemma 6.4.7 Let α be a data space containing volatile information. The priority of the α-overrun handler must be higher than the priorities of all α overrun dependent modules which could potentially be active when an overrun on α is detected.

Since overrun handlers may have a rather high priority, they must be elaborated in a very small real-time interval, otherwise the detection of an overrun may cascade into the detection of several different overruns. Let us examine then the actions which must be performed by an overrun handler.

It should be noted that overruns not necessarily require a major decision effort in order to be serviced. For example, when an external interrupt is generated at a terminal, all further external interrupts may be disregarded until the first of these external interrupts is completely serviced. It should be clear that this "dropping" of exceptions can be accomplished by means of a linkage definition linking a "dummy" handler to the qualifying exception descriptor. This dummy handler is still elaborated with a higher priority, however its existence is almost non noticeable.

The occurrence of an overrun could also signal that the exception handler servicing the previous exception is to be simply restarted. For example, an airplane tracking radar system could have been designed in such a way that the successive positions of the airplanes are constantly being predicted. At given points in time these predictions are corrected according to data obtained from the radar. Now, these corrections could be activated by means of some exception, e.g. at a constant time interval. Since failing

to perform the corrections for a few consecutive cycles does not endanger the performance of the system as a whole, an overrun could safely be allowed to restart the correction algorithm with the new data, possibly with an increased priority.

The overrun handling can be viewed as consisting of two actions, one of which preempts all necessary modules [theorem 6.4.6], and the other which effectively services the overrun signal. This follows from the fact that, once a module instance has been preempted due to an exception signal, the elaboration of this module instance can only be resumed by means of handler action. Thus, if the preempt exception handler does not provide for the resumption of the module instance elaboration, this resumption can only be achieved by the overrun_service exception handler. We have then:

Fact 6.4.8 Let α be a data space containing volatile information. An overrun signal on α can be broken into two signals:

i- a preempt signal which corresponding handler must preempt all necessary modules currently being elaborated;

ii- an overrun_service signal which handler effectively services the overrun exception.

Now, if the preempt exception handler is not hardware implemented, it may take some time until all necessary modules are effectively preempted. It follows then, that in some modules meaningless elaboration has already been performed when these modules are actually preempted. Now, if these modules are able to produce side effects, e.g.

output operations, the degree of damage due to a data space overrun may be unknown. We have then:

Fact 6.4.9 Let α be a data space containing volatile information. Let M be the set of module instances which are being elaborated when an overrun, i.e. preempt, on α is detected. In order to keep the overrun damage local, all side effect generating operations within the module instances of the set M must be delayed until the preempt exception handler unmasks.

Fact 6.4.9 tells us that all store and output operations must be delayed within all processors which could possibly be elaborating a module which is overrun dependent on the data space overrun. Thus, a preempt signal is a signal which must be transmitted immediately to all of a selected group of processors in a multiprocessor machine. This could be implemented by having a "store inhibit" linked to all storage devices. This store inhibit is active whenever one of the several processors is in the preempt state. Thus any write access requested by any of the processors, except for the processor in the preempt state, will delay these processors until the store inhibit is reset.

Although the cost of overrun damage may be quite small in many cases, in some cases the results could be disastrous. Consider for instance a process controller which steers a missile to a target detected by a radar. If the radar position generator function causes an overrun, and the incorrect position is fed to the missile steering module, the target could be missed and elaboration could terminate in real disaster.

7. Epilogue.

Having read through this rather lengthy dissertation, one is faced with the question: was it worthwhile? Let us provide then a summary of what we consider being the most important points treated:

i- the author has been unable to find a text in the current literature which studies the language design problems relative to modularity and intra-module communication in a unified form.

ii- with few exceptions, e.g. clusters as defined by Liskov and Zilles[Lis1], none of the existing or proposed programming languages treats type descriptors as values, being then unable to perform type descriptor transmission. As has been shown in section 3.3, the ability of transmitting type descriptors considerably increases the flexibility of modules. The type descriptors we have introduced here are more general than the clusters of Liskov and Zilles, since our type descriptors can be any kind of module, possibly even a coroutine as required for generator functions.

iii- none of the existing or proposed programming languages provides means to define explicitly a controlled environment for reference typed values, i.e. data space identifications. As has been shown in section 4.3, data space identifications enable us to link a reference typed value both to a target type and to a restricted set of access functions. This implies that the access paths are well defined and supervised.

iv- with few exceptions, e.g. SNOBOL4, none of the existing or proposed languages allows us to interfere with language provided operations. As has been shown in section 4.4 the ability of defining O-emulating functions allows us to establish a tight control on how data space identifications can be disseminated and on what modules could access a given piece of information.

v- with few exceptions, e.g. IPL-V, none of the proposed or existing programming languages provides means to explicitly define generator functions. As should be evident from the discussion of section 4.5, generator functions aid considerably in hiding the implementational information of modules. They allow also the "modularization" of operations which require a coroutine environment in order to be implemented, e.g. tree traversal operations.

vi- few of the existing or proposed programming languages address themselves to unordered parameter lists. As has been shown in section 5.2, the named association rule aids in making parameter list association clearer and provides means to control the meaningfulness of parameter association.

vii- module typed variables have received little attention so far. It is usually forbidden to consider arrays of modules or even of modules which return module typed values, although not in LISP 1.5 or GEDANKEN. In section 5.3 we have shown that there is no major reason for such restrictions. Furthermore, we have shown that, by means of piecemeal associating parameters, the flexibility of modules is increased.

Finally, due to the named association rule, there are no more restrictions on the lengths of parameter lists to be associated. Thus, when using default parameter values, modules can be used without having their standard environment requirements defined over and over again.

viii- exception handling is not present or is not satisfactorily dealt with in existing or proposed programming languages. Except for few isolated studies addressing themselves to particular problems[Elm1,Hor1,Con1, Zil2], no thorough study of the necessary tools has yet been presented.

B.1 Abbreviations.

ACM	Association for Computing Machinery
Acta Inf	Acta Informatica
AFIPS	American Federartion of Information Processing Societies -SJCC - Spring Joint Computer Conference -FJCC - Fall Joint Computer Conference
ANSI	American National Standard Institute
BIT	Nordisk Tidskrift for Informationsbehandling
CACM	Communications of the ACM
CJ	The Computer Journal
CS-ACM	Computing Surveys of the ACM
IAGJ	IAG Journal
IBM-SJ	IBM Systems Journal
IEEE	Institute of Electric and Electronic Engineers
IJCIS	International Journal of Computer and Information Sciences
IPL	Information Processing Letters
JACM	Journal of the ACM
JCSS	Journal of Computer and Systems Sciences
Num Math	Numerische Mathematic
SIGPLAN	Special Interest Group on Programming Languages, ACM

SPE Software Practice and Experience

B.2 Bibilography Listing.

[And1] Anderson, J.P.
'Program Structures for Parallel Processing'; CACM vol 8; no 12; Dec 1965; pp 786-788

[ANS1] ANSI
ANSI Standard FORTRAN; ANSI; Oct 1969

[Ash1] Ashcroft, E.A.
'Program Correctness Methods and Language Definition'; SIGPLAN vol 7; no 1; Jan 1972; pp 51-57

[Ayr1] Ayres, R.B.; Derrenbacher, R.L.
'Partial Recompilation'; AFIPS-SJCC vol 38; 1971; pp 487-502

[Bae1] Baecker, H.D.
'Garbage Collection for Virtual Memory Computer Systems'; CACM vol 15; no 11; Nov 1972; pp 981-985

[Bae2] Baecker, H.D.
'Implementing the Algol 68 Heap'; BIT vol 10; 1970; pp 405-414

[Bak1] Baker, J.L.
'An Unintentional Omission from Algol 68'; IPL vol 1; no 6; Dec 1972; pp 244-245

[Bal1] Balzer, R.M.
'PORTS - a Method for Dynamic Interprogram Communication and Job Control'; AFIPS-SJCC vol 38; 1971; pp 485-490

[Bar1] Barron, D.W.
Assemblers and Loaders; MacDonald, American Elsevier Inc.; 1969

[Bar2] Barron, D.W.; Buxton, J.N.; Hartley, D.F.; Nixon, E.; Strachey, C.
'The Main Features of CPL'; CJ vol 6; no 2; Jul 1963; pp 134-143

[Bar3] Barron, D.W.; Jackson, I.R.
'The Evolution of Job Control Languages'; SPE vol 2; no 2; Apr 1872; pp 143-164

[Bau1] Bauer, A.M.; Saal, H.J.
'Does APL Really Need Run Time Checking?'; SPE vol 4; no 2; 1874; pp 129-138

[Bau2] Bauer, F.L. ed.
Advance Course on Software Engineering; Lecture Notes in Economics and Mathematics 81; 1973

[Bau3] Bauer, H.R.
Introduction to ALGOLW Programming; Computer Science Department, Stanford University; Jul 1969

[Bau4] Baumann, R.; Feliciano, M.; Bauer, F.L.; Samelson, K.
Introduction to Algol 60; Englewood Cliffs, Prentice Hall; 1964

[Bay1] Bayer, R.; Gries, D.; Paul, M.; Wiehle, H.R.
'The ALCOR ILLINOIS 7090/7094 Post Mortem Dump'; CACM vol10; no 12; Dec 1967; pp 804-808

[Bay2] Bayer, R.; Murphree, E.; Gries, D.
User's Manual for the ALCOR ILLINOIS 7090 ALGOL60 Translator. secon edition; University of Illinois; Sep 1964

[Ben1] Bennet, R.K.; Neumann, H.D.
'Extension of Existing Compilers by Sophisticated use of Macros'; CACM vol 7; no 9; Sep 1964; pp 541-542

[Ben2] Bensoussan, A.; Clingen, C.T.; Daley, R.C.
'The MULTICS Virtual Memory: Concepts and Design'; CACM vol 15; no 5; May 1972; pp 308-318

[Ber1] Berry, D.M.
'The Importance of Implementation Modules in Algol 68'; SIGPLAN vol 5; no 9; Sep 1970; pp 14-24

[Ber2] Berry, D.M.
'Introduction to Oregano'; in Toui Wegner eds.; Feb 1971; pp 171-190

[Ber3] Berry, D.M.
On the Design and Specification of the Programming Language OREGANO; University of California Los Angeles, Computer Science Department, UCLA-ENG-7388; Jan 1974

[Ber4] Berry, D.M.; Chirica, L.; Johnston, J.B.; Martin, D.F.; Sorkin, A.
On the Time Required for Retention; University of California Los Angeles, Computer Science Department, Modelling and Measurement Note no 20; Oct 1973

[Bob1] Bobrow, D.G. ed.
Symbol Manipulation Languages and Techniques; North Holland; 1968

[Bob2] Bobrow, D.G.; Buchfield, J.D.; Murphy, D.L.; Tomlinson, R.S.
'Tenex, a Paged Time-Sharing System for the PDP-10'; CACM vol 15; no 3; Mar 1972; pp 135-143

[Bob3] Bobrow, D.G.; Wegbreit, B.
'A Model and Stack Implementation of Multiple Environments'; CACM vol 16; no 10; Oct 1973; pp 591-603

[Bra1] Branquart, P.; Lewi, J.
'A Scheme of Storage Allocation and Garbage Collection for Algol 68'; in Peck ed.; 1970; pp 199-238

[Bra2] Branquart, P.; Lewi, J.; Sintzoff, M.; Wodon, P.L.
'The Composition of Semantics in Algol 68'; CACM vol 14; no 11; Nov 1971; pp 697-708

[Bro1] Brown, P.J.
'Levels of Language for Portable Software'; CACM vol 15; no 12; Dec 1972; pp 1059-1062

[Bro2] Brown, W.S.
'An Operating Environment for Dynamic Recursive Computer Programming Systems'; CACM vol 8; no 6; Jun 1965; pp 371-377

[Bro3] Brown, W.S.
A Tutorial Guide to ALTRAN; Bell Telephone Laboratories, Murray Hill, N.J.; 1970

[Bro4] Brown, W.S.
ALTRAN User's Manual, Third Edition; Bell Telephone Laboratories, Murray Hill, N.J.; 1973

[Bur1] Burroughs
Burroughs B6500 Reference Manual; Burroughs Corporation; Detroit, Michigan; 1969

[Cla1] Clark, B.L.; Horning, J.J.
'The System Language for Project SUE'; SIGPLAN vol 6; no 9; Oct 1971; pp 79-85

[Cli1] Clint, M.
'Program Proving: Coroutines'; Acta Inf vol 2; fasc 1; 1973; pp 50-63

[Cli2] Clint, M.; Hoare, C.A.R.
'Program Proving: Jumps and Functions'; Acta Inf vol 1; fasc 3; 1972; pp 214-224

[Coc1] Cocke, J.; Schwartz, J.T.
Programming Languages and Their Compilers; Courant Institute of Mathematical Sciences, New York University, Second Revised Edition; Apr 1970

[Cof1] Coffman, E.G.; Denning, P.J.
Operating Systems Theory; Prentice Hall; 1973

[Cof2] Coffman, E.G.; Elphick, M.J.; Shoshani, A.
'System Deadlocks'; CS-ACM vol 3; no 2; Jun 1971; pp 67-78

[Coh1] Cohen, J.; Trilling, L.
'Remarks on Garbage Collection using a Two Level Storage'; BIT vol 7; 1967; pp 22-30

[Con1] Connet, J.R.; Pasternack, E.J.; Wagner, B.D.
'Software Defenses in Real-time Control Systems'; IEEE International Symposium on Fault Tolerant Computing; 1972; pp 94-99

[Con2] Conway, M.E.
'Design of a Separable Transition Diagram Compiler'; CACM vol 6; no 7; Jul 1963; pp 396-408

[Dah1] Dahl, O.J.; Dijkstra, E.W.; Hoare, C.A.R. eds.
Structured Programming; Academic Press, London, New York; 1972

[Dah2] Dahl, O.J.; Hoare, C.A.R.
'Hierarchical Program Structures'; in Dahl, Dijkstra and Hoare eds.; 1973; pp 175-220

[Dah3] Dahl, O.J.; Myhrhaug, B.; Nygaard, K.
Simula 67 Common Base Language; Norwegian Computing Centre, Publication S-22; Oct 1972

[Dah4] Dahl, O.J.; Nygaard, K.
'Simula - an Algol Based Simulation Language'; CACM vol 9; no 9; Oct 1966; pp 671-678

[Dal1] Daley, R.G.; Dennis, J.B.
'Virtual Memory, Processes and Sharing in MULTICS'; CACM vol 11; no 5; May 1968; pp 306-312

[Dav1] Davis, A.L.
SPL = A Structured Programming Language; Dept. of Electrical Engineering, University of Utah; Sep 1972

[DEC1] DEC
Program Logic Manual for the PDP-10 Time Sharing Monitor; Digital Equipment Corporation; Jan 1969

[DEC2] DEC
PDP-10 Reference Handbook; Digital Equipment Corporation; 1969

[Den1] Denning, P.J.
'Virtual Memory'; CS-ACM vol 2; no 3; Sep 1970; pp 153-189

[Den2] Dennis, J.B.
'On the Design and Specification of a Common Base Language'; Symposium on Computers and Automata, Polytechnic Institute of Brooklyn; Apr 1971

[Den3] Dennis, J.B.
'Modularity'; in Bauer ed.; 1973; pp 128-182

[Des1] Desjardins, P.
'Dynamig Data Structure Mapping'; SPE vol 4; no 2; 1974; pp155-162

[Dew1] Dewar, R.B.K.
SPITBOL; Illinois Institute of Technology; Feb 1971

[Dij1] Dijkstra, E.W.
'Recursive Programming'; Num Math vol 2; 1960; pp 312-318

[Dij2] Dijkstra, E.W.
'Solution of a Problem in Concurrent Programming Control'; CACM vol 8; no 9; Sep 1965; pg 569

[Dij3] Dijkstra, E.W.
'Cooperating Sequential Processes'; in Genuys ed.; 1968; pp 43-112

[Dij4] Dijkstra, E.W.
'A Constructive Approach to the Problem of Program Correctness'; BIT vol 8; 1968; pp 174-186

[Dij5] Dijkstra, E.W.
A Short Introduction to the Art of Computer Programming; Technische Hogeschool, Eindhoven, Report EWD 316; 1971

[Dij6] Dijkstra, E.W.
'Hierarchical Ordering of Sequential Processes'; Acta Inf vol 1; fasc 2; 1971; pp 115-138

[Dij7] Dijkstra, E.W.
'A Class of Allocation Strategies Inducing Bounded Delays only'; AFIPS-SJCC vol 40; 1972; pp 933-936

[Dij8] Dijkstra, E.W.
A Simple Axiomatic Basis for Programming Language Constructs; Technische Hogeschool, Eindhoven, Report EWD 372-0; 1973

[Dij9] Dijkstra, E.W.
'Notes on Structured Programming'; in Dahl; Dijkstra, Hoare eds.; 1973; pp 1-82

[Ear1] Earley, J.
'Towards an Understanding of Data Structures'; CACM vol 14; no 10; Oct 1971; pp 617-627

[Ear2] Earley, J.
'Relational Level Data Structures for Programming Languages'; Acta Inf vol 2; fasc 4; 1973; pp 293-310

[Ear3] Earley, J.
'High Level Operations in Automatic Programming'; SIGPLAN vol 9; no 4; Apr 1974; pp 34-42

[Ear4] Earley, J.; Caizergues, P.
'A Method for Incrementally Compiling Languages with Nested Statement Structure'; CACM vol 15; no 12; Dec 1972; pp 1040-1044

[Elm1] Elmendorf, W.R.
'Fault Tolerant Programming'; IEEE Fault Tolerant Computing Symposium; Jun 1972; pp 79-83

[Elm2] Elmendorf, W.R.
'Disciplined Software Testing'; in Rustin ed.; 1970; pp 137-139

[Els1] Elshoff, J.L.; Beckermeyer, R.; Dill, J.; Marcotty, M.; Murray, J.
'Handling Assynchronous Interrupts in a PL/1-like Language'; SPE vol 4; no 2; 1974; pp 117-124

[Els2] Elspas, B.; Levitt, K.N.; Waldinger, R.J.; Waksman, A.
'An Assessment of Techniques for Proving Program Correctness'; CS-ACM vol 4; no 2; Jun 1972; pp 97-147

[Emd1] van Emden, M.H.
'Hierarchical Decomposition of Complexity'; Machine Intelligence vol 5; 1970; pp 361-380

[Ers1] Ershov, A.P.
'A Multilanguage Programming System oriented to Language Description and Universal Optimization Algorithms'; in Peck ed.; 1970; pp 143-162

[Eve1] Evershed, D.G.; Rippon, G.E.
'High Level Languages for Low Level Users'; CJ vol 14; no 1; Feb 1971; pp 87-90

[Fel1] Feldman, J.A.; Rovner, P.D.
'An Algol Based Associative Language'; CACM vol 12; no 8; Aug 1969; pp 439-449

[Fen1] Fenichel, R.R.
'On Implementation of Label Variables'; CACM vol 14; no 5; May 1971; pp 348-350

[Fis1] Fischer, A.E.; Fischer, M.J.
'Mode Modules as Representation of Domains'; ACM Symposium on Principles of Programming Languages; Oct 1973; pp 139-143

[Fis2] Fisher, D.
Control Structures for Programming Languages; Carnegie Mellon University; 1970

[Fle1] Fleck, A.
'Towards a Theory of Data Structures'; JCSS vol 5; no 5; Oct 1971; pp 475-488

[Flo1] Floyd, R.W.
'Assigning Meanings to Programs'; Proceedings of a Symposium in Applied Mathematics vol 19; Mathematical Aspects of Computer Science; American Mathematical Society; 1967; pp 19-32

[Fra1] Fraser, A.G.
'On the Meaning of Names in Programming Systems'; CACM vol 14; no 6; Jun 1971; pp 409-416

[Gel1] Gelernter, H.; Hansen, J.R.; Gerberich, C.L.
'A FORTRAN Compiled List Processing Language'; JACM vol7; no 1; Apr 1960; pp 87-101

[Gen1] Gentleman, W.M.
'A Portable Coroutine System'; IFIP Computer Software Booklet TA-3; Aug 1971; pp 94-98

[Gen2] Genuys, F. ed.
Programming Languages; Academic Press; 1968

[Gim1] Gimpel, J.F.
A Design for SNOBOL4 for the PDP 10, Part I - the General; Bell Telephone Laboratories Inc., Holmdel, New Jersey, S4D29b; May 1973

[Gim2] Gimpel, J.F.
SITBOL Version 3.0; Bell Telephone Laboratories Inc., Holmdel, New Jersey, S4D30b; Jun 1973

[Gna1] Gnatz, R.
Sets and Predicates in Programming Languages; International Summer School on Structured Programming and Programmed Structures, Munich, Germany; Aug 1973

[Gol1] Goldberg, P.C.; Goldberg, R.
Some Remarks on Programming Language Design; IBM Research Report, RC 3458; Jul 1971

[Gol2] Goldsmith, C.W.
'The Design of a Procedureless Programming Language'; SIGPLAN vol 9; no 4; Apr 1974; pp 13-24

[Goy1] Goyer, P.
'A Garbage collector to be Implemented on a CDC 3100'; in Peck ed.; 1970; pp 303-320

[Gra1] Graham, R.M.; Schroeder, M.D. eds.
Programming Languages = Operating Systems; SIGPLAN vol 8; no 9; Sep 1973

[Gri1] Gries, D.
Compiler Construction for Digital Computers; John Wiley, Sons; 1971

[Gri2] Gries, D.; Paul, M.; Wiehle, H.R.
'Some Techniques Used in the ALCOR ILLINOIS 7090'; CACM vol 8; no 8; Aug 1965; pp 496-500

[Gri3] Gries, D.; Paul, M.; Wiehle, H.R.
ALCOR ILLINOIS 7090- An ALGOL Compiler for the 7080; Rechenzentrum der Technischen Hochschule München, Report no 6415; 1964

[Gri4] Griswold, R.E.
'Suggested Revisions and Additions to the Syntax and Control Mechanisms of SNOBOL4'; SIGPLAN vol 9; no 2; Feb 1974; pp 7-23

[Gri5] Griswold, R.E.
The Macro Implementation of SNOBOL4; W.H. Freeman and Co.; 1972

[Gri6] Griswold, R.E.; Poage, J.F.; Polonsky, I.P.
The SNOBOL4 Programming Language; Prentice Hall Inc., Second Edition; 1971

[Gur1] Gurski, A.
'Job Control Languages as Machine Oriented Languages'; SIGPLAN vol 8; no 3; Mar 1973; pp 18-23

[Hab1] Habermann, A.N.
'Prevention of System Deadlocks'; CACM vol 12; no 7; Jul 1969; pp 373-377, 385

[Hab2] Habermann, A.N.
'Critical Comments on the Programming Language PASCAL'; Acta Inf vol 3; fasc 1; 1973; pp 47-57

[Hai1] Haines, E.C.
'AL: A Structured Assembly Language';
SIGPLAN vol 8, no 1; Jan 1973; pp 15-20
SIGPLAN vol 8, no 4; Apr 1973; pp 16-21

[Hal1] Hall, A.D.
'The ALTRAN System for Rational Function Manipulation - A Survey'; CACM vol 14; no 8; Aug 1971; pp 517-521

[Han1] Haney, F.M.
'Module Connection Analysis - A Tool for Scheduling Software Debugging Activities'; AFIPS-FJCC vol 41-1; 1972; pp 173-179

[Han2] Hansen, P.B.
Concurrent Programming Concepts; International Summer School on Structured Programming and Programmed Structures, Munich, Germany; Aug 1973

[Han3] Hansen, P.B.
Operating System Principles; Prentice Hall; 1973

[Har1] Harrison, M.C.
Data Structures, Programming; Courant Institute of Mathematical Sciences, New York University; 1971

[Har2] Harrison, W.
The Production of Canonical Forms for Data Type Representations; IBM Research Report, RC 4420; Jul 1973

[Hat1] Hatfield, D.J.; Gerald, J.
'Program Restructuring for Virtual Memory'; IBM-SJ vol 10; no 3; 1971; pp 168-192

[Hau1] Hauck, E.A.; Dent, B.A.
'Bouroughs B6500/B6700 Stack Mechanism'; AFIPS-SJCC vol 32; 1968; pp 245-251

[Hen1] Henhapl, W.; Jones, C.B.
'A Run Time Mechanism for Referencing Variables'; IPL vol 1; no 1; 1971; pp 14-16

[Hir1] Hirschberg, D.S.
'A Class of Dynamic Memory Allocation Algorithms'; CACM vol 16; no 10; Oct 1973; pp 615-618

[Hoa1] Hoare, C.A.R.
'Record Handling'; in Genuys ed.; 1968; pp 291-348

[Hoa2] Hoare, C.A.R.
'An Axiomatic Basis for Computer Programming'; CACM vol 12; no 10; Oct 1969; pp 576-580, 583

[Hoa3] Hoare, C.A.R.
'Procedures and Parameters: an Axiomatic Approach'; in Engeler ed.; Oct 1970; pp 102-115

[Hoa4] Hoare, C.A.R.
'Proof of Correctness of Data Representations'; Acta Inf vol 1; 1972; pp 271-281

[Hoa5] Hoare, C.A.R.
'Notes on Data Structuring'; in Dahl, Dijkstra and Hoare eds.; 1973; pp 83-174

[Hoa6] Hoare, C.A.R.
'A Note on the FOR Statement'; BIT vol 12; 1972; pp 347-365

[Hoa7] Hoare, C.A.R.
'Hints on Programming Language Design'; Invited Address at SIGACT/SIGPLAN Symposium on Principles of Programming Languages; Oct 1973

[Hoa8] Hoare, C.A.R.
'Towards a Theory of Parallel Programming'; in Hoare; Perrot eds.; Sep 1971; pp 61-71

[Hoa9] Hoare, C.A.R.; Perrot, R.M. eds.
Operating Systems Techniques; Academic Press; 1972

[Hoa10] Hoare, C.A.R.; Wirth, N.
'An Axiomatic Definition of the Programming Language PASCAL'; Acta Inf vol 2; fasc 4; 1973; pp 335-356

[Hol1] Holt, R.C.
'Comments on Prevention of Systems Deadlocks'; CACM vol 14; no 1; Jan 1971; pp 36-38

[Hol2] Holt, R.C.
On Deadlock in Computer Systems; Cornell University; Jan 1971

[Hol3] Holt, R.C.
'Some Deadlock Properties of Computer Systems'; CS-ACM vol 4; no 3; Sep 1972; pp 179-196

[Hon1] Honeywell
Series 6000 Summary Description; Honeywell Information Systems, DA48; 1972

[Hon2] Honeywell
Series 600/6000 Macro Assembler Program (GMAP); Honeywell Information Systems, BN86; 1973

[Hor1] Horning, J.J.; Lauer, H.C.; Melliar-Smith, P.M.; Randell, B.
A Program Structure for Error Detection and Recovery; University of New Castle Upon Tyne, Technical Report Series, no 59; Apr 1974

[IBM1] IBM
IBM System /360 Principles of Operation; IBM Corp., GA22-6821; Sep 1968

[IBM2] IBM
PL/1 Language Specifications; IBM GY33-6003-2; Jun 1970

[IBM3] IBM
IBM System /360 Operating System. PL/1 (F). Language Reference Manual; IBM GC28-8201-3, Fourth Edition; Jun 1970

[IBM4] IBM
IBM System/360 Operating System Assembler Language; IBM GC28-6514 sixth edition; Jun 1969

[Ing1] Ingerman, P.Z.
'THUNKS A Way of Compiling Procedure Statements with some Comments on Procedure Declarations'; CACM vol 4; no 1; Jan 1961; pp 55-58

[Ive1] Iverson, K.E.
A Programming Language; John Wiley and Sons; 1962

[Joh1] Johnson, S.C.; Kerningham, B.W.
The Programming Language B; Bell Telephone Laboratories, Murray Hill, N.J. Computing Science Technical Report #8; Jan 1973

[Joh2] Johnston, J.B.
'The Contour Model of Block Structured Processes'; in Toui Wegner eds.; Feb 1971; pp 55-82

[Kai1] Kain, R.Y.
'Block Structures, Indirect Addressing and Garbage Collection'; CACM vol 12; no 7; Jul 1969; pp 395-398

[Kat1] Katzan, H.
'An APL Approach to the Representation and Manipulation of Data Structures'; IJCIS vol 1; no 2; Jun 1972; pp 93-113

[Ker1] Kerningham, B.W.; Plauger, P.J.
Programming Style for Programmers and Language Designers; Bell Laboratories, Murray Hill, New Jersey; 1973

[Kno1] Knowlton, K.C.
'A Fast Storage Allocator'; CACM vol 8; no 10; Oct 1965; pp 623-625

[Kno2] Knowlton, K.C.
'A Programmers Description of L^6'; CACM vol 9; no 8; Aug 1966; pp 616-625

[Knu1] Knuth, D.E.
The Art of Computer Programming
Vol 1 Fundamental Algorithms; 1968
Vol 2 Seminumerical Algorithms; 1970
Vol 3 Sorting and Searching; 1972
Addison Wesley

[Knu2] Knuth, D.E.
'An Empirical Study of FORTRAN Programs'; SPE vol 1; no 2; Apr 1971; pp 105-133

[Knu3] Knuth, D.E.
A Review of "Structured Programming"; Stanford University, Computer Science Department, STAN-CS-73-371; Jun 1973

[Knu4] Knuth, D.E.; Bumgarner, L.L.; Ingerman, P.Z.; Merner, J.N.; Hamilton, D.E.; Lietzke, M.P.; Ross, D.T.
'A Proposal for Input Output Conventions in Algol 60'; CACM vol 7; no 5; May 1964; pp 273-283

[Knu5] Knuth, D.E.; Merner, J.N.
'Algol 60 Confidential'; CACM vol 1; no 6; Jun 1961; pp 268-272

[Kos1] Kosinski, P.R.
A Data Flow Language for Operating Systems Programming; IBM Research Report RC 4166; Dec 1972

[Kos2] Kosinski, P.R.
A Data Flow Programming Language; IBM Research Report RC 4264; Mar 1973

[Lae1] Laevenworth, B.M. ed.
Special Issue on Control Structures in Programming Languages; SIGPLAN vol 7; no 11; Nov 1972

[Lam1] Lampson, B.W.
'A Note on the Confinement Problem'; CACM vol 16; no 10; Oct 1973; pp 613-615

[Lan1] Langmaack, H.
'On Procedures as Open Subroutines I'; Acta Inf vol 2; fasc 4; 1973; pp 311-334

[Led1] Ledgard, H.F.
'Ten Mini Languages: A Study of Topical Issues in Programming Languages'; CS-ACM vol 3; no 3; Sep 1971; pp 115-146

[Led2] Ledgard, H.F.
'A Model for Type Checking'; CACM vol 15; no 11; Nov 1972; pp 956-966

[Lee1] Lee, J.A.N.
'The Formal Definition of the BASIC Language'; CJ vol 15; no 1; Feb 1972; pp 37-41

[Lew1] Lewis, C.H.; Rosen, B.K.
'Recursively Defined Data Types'; ACM Symposium on Principles of Programming Languages; Oct 1973; pp 125-138

[Lin1] Lindsey, C.H.; van der Meulen, S.G.
Informal Introduction to Algol 68; North Holland Publishing Co.; 1971

[Lip1] Lipovski, G.J.
'On Data Structures in Associative Memories'; in Tou; Wegner eds.; Feb 1971; pp 346-365

[Lis1] Liskov, B.; Zilles, S.N.
'Programming with Abstract Data Types'; SIGPLAN Symposium on Very High Level Languages; Mar 1974; pp 50-59

[Lum1] Lum, V.Y.
'General Performance Analysis of Key-to-Address Transformation Methods Using an Abstract File Concept'; CACM vol 16; no 10; Oct 1973; pp 603-612

[Mad1] Madnick, S.E.
'A Modular Approach to File System Design'; IAGJ vol 2; no 3; 1969; pp 7-34

[Mar1] Marshall, S.
'An Algol 68 Garbage Collector'; in Peck ed.; 1970; pp 239-244

[McC1] McCarthy, J.; Abrahams, P.W.; Edwards, D.J.; Hart, T.P.; Levin, M.I.
LISP 1.5 Programmer's Manual; MIT Press; 1962

[McC2] McCarthy, J.; Corbató, F.J.; Daggett, M.M.
'The Linking Segment Subprogram Language and Linking Loader'; CACM vol 6; no 7; Jul 1963; pp 391-395

[McK1] McKeeman, W.M.; Horning, J.J.; Wortman, D.B.
A Compiler Generator; Prentice Hall Inc.; 1970

[Mil1] Miller, J.S.; Vandever, W.H.
'On Software Quality'; IEEE Fault Tolerant Computing Symposium; Jun 1972; pp 84-88

[Mor1] Morris, J.H.
'Protection in Programming Languages'; CACM vol 16; no 1; Jan 1973; pp 15-21

[Mor2] Morris, J.H.
'Types are not Sets'; ACM Symposium on Principles of Programming Languages; Oct 1973; pp 120-124

[Sho1] Shooman, M.L.
'Probabilistic Methods for Software Reliability Prediction'; IEEE Fault Tolerant Computing Symposium; Jun 1972; pp 211-215

[Sib1] Sibley, E.H.; Taylor, R.W.
A Data Definition and Mapping Language; Michigan University Ann Arbor, Department of Industrial and Operations Engineering, AFOSR-TR-73-1521; Jan 1972

[Sim1] Simon, J.C.; Guiho, G.
'On Algorithms Preserving Neighborhood to File and Retrieve Information in a Memory'; IJCIS vol 1; no 1; Mar 1972; pp 3-15

[Sit1] Sites, R.L.
ALGOLW Reference Manual; Stanford University; Feb 1972

[Slu1] Slutz, D.R.
The Flow Graph Schemata Model of Parallel Computation; MIT Project MAC, MAC-TR-53; Sep 1968

[Smi1] Smith, D.C.; Enen, H.J.
MLISP2; Stanford University, Computer Science Department, STAN-CS-73-356; May 1973

[Smi2] Smith, D.K.
'An Introduction to the List Processing Language SLIP'; in Rosen ed.; 1967; pp 393-418

[Sol1] Solutseff, N.
'A Classification of Extensible Programming Languages'; IPL vol 1; no 3; 1972; pp 91-96

[Sol2] Solutseff, N.
'On a Notational Device for the Description of Pointer Free Operations on Structured Data'; IPL vol 2; no 6; Apr 1974; pp 158-159

[Sta1] Standish, T.A.
A Data Definition Facility for Programming Languages; Carnegie Institute of Technology; May 1967

[Tho1] Thorelli, L.E.
'Marking Algorithms'; BIT vol 12; 1972; pp 555-568

[Tou1] Tou, J.T.; Wegner, P. eds.
Proceedings of a Symposium on Data Structures in Programming Languages; SIGPLAN vol 6; no 2; Feb 1971

[Tur1] Tursky, W.M.
'Data Structures and Their Ordering'; IAGJ vol 3; no 2; 1970; pp 141-150

[Tur2] Tursky, W.M.
'A Model for Data Structures and its Applications';
Acta Inf vol 1; fasc 1; 1971; pp 26-34
Acta Inf vol 1; fasc 4; 1972; pp 282-289

[Wan1] Wang, A.; Dahl, O.J.
'Coroutine Sequencing in a Block Structured Environment'; BIT vol 11; 1971; pp 425-449

[Wat1] Watson, R.W.
Time-sharing System Design Concepts; McGraw Hill Book Co.; 1970

[Weg1] Wegner, P.
Programming Languages, Information Structures and Machine Organization; McGraw Hill, New York; 1968

[Weg2] Wegner, P.
'Data Structure Models for Programming Languages'; in Tou; Wegner eds.; Feb 1971; pp 1-54

[Wei1] Weissman, C.
LISP 1.5 Primer; Dickenson Publishing Co. Inc.; 1967

[Wei2] Weizenbaum, J.
'Symmetric List Processor'; CACM vol 6; no 9; Sep 1963; pp 524-544

[Wel1] Wells, M.B.; Morris, J.B.
'The Unified Data Structure Capability in MADCAP VI'; IJCIS vol 1; no 3; Sep 1972; pp 193-208

[Wha1] Whaley, A.D.
'A Failure Tolerant Filing System'; SPE vol 2; no 3; Jul 1972; pp 287-291

[Wil1] Wile, D.S.; Geschke, C.
'An Implementation Base for Efficient Data Structuring'; IJCIS vol 1; no 3; Sep 1972; pp 209-224

[Wil2] Wilkes, M.V.
'Associative Tabular Data Structures'; IJCIS vol 1; no 3; Sep 1972; pp 225-233

[Wil3] Williams, R.
'A Survey of Data Structures for Computer Graphics Systems'; CS-ACM vol 3; no 1; Mar 1971; pp 1-21

[Win1] van Winjgaarden, A.; Mailloux, B.J.; Peck, J.E.L.; Koster, C.H.A.
'Report on the Algorithmic Language Algol 68'; Num Math vol 14; 1969; pp 79-218

[Wir1] Wirth, N.
'PL360, a Programming Language of the 360 Computers'; JACM vol 15; no 1; Jan 1968; pp 37-74

[Wir2] Wirth, N.
'The Programming Language PASCAL'; Acta Inf vol 1; 1971; pp 35-63

[Wir3] Wirth, N.
'Program Development by Stepwise Refinement'; CACM vol 14; no 4; Apr 1971; pp 221-227

[Wir4] Wirth, N.
The Programming Language PASCAL (Revised Report); Eidgenössische Hochschule, Zürich, CSTR 5; Nov 1972

[Wir5] Wirth, N.
Systematic Programming; Prentice Hall; 1973

[Wir6] Wirth, N.; Hoare, C.A.R.
'A Contribution to the Development of Algol'; CACM vol 9; no 6; Jun 1966; pp 413-431

[Wir7] Wirth, N.; Weber, H.
'EULER: A Generalization of Algol and its Formal Definition'
CACM vol 9; no 1; Jan 1966; pp 13-23, 25
CACM vol 9; no 2; Feb 1966; pp 89-99

[Wod1] Wodon, P.L.
'Data Structures and Storage Allocation'; BIT vol 9; 1969; pp 270-282

[Wod2] Wodon, P.L.
'Methods of Garbage Collection for Algol 68'; in Peck ed.; 1970; pp 245-264

[Woo1] Woodward, P.M.
'Practical Experience with Algol 68'; SPE vol 2; no 1; 1972; pp 7-19

[Wul1] Wulf, W.A.; Russell, D.B.; Habermann, A.N.
'BLISS: A Language for Systems Programming'; CACM vol 14; no 12; Dec 1971; pp 780-790

[Wul2] Wulf, W.A.; Russell, D.B.; Habermann, A.N.; Geschke, C.; Apperson, J.; Wile, D.S.
BLISS Reference Manual; Carnegie Mellon University, Computer Science Dept.; Jan 1970

[Wul3] Wulf, W.; Shaw, M.
'Global Variable Considered Harmful'; SIGPLAN vol 8; no 2; Feb 1973; pp 28-34

[You1] Youngs, E.A.
Error Proneness in Programming; University of North Carolina; 1970

[Zel1] Zelkowitz, M.
'Interrupt Driven Programming'; CACM vol 14; no 6; Jun 1971; pp 417-418

[Zil1] Zilles, S.N.
'Procedural Encapsulation: A Linguistic Protection Technique'; in Graham, Schroeder eds.; Sep 1973; pp 142-146

[Zil2] Zilles, S.N.
Working Notes on Error Handling; MIT Project MAC, CLU Design Note 6; Jan 1974