MIT/LCS/TR-234

TRANSMITTING ABSTRACT VALUES IN MESSAGES

Maurice Peter Herlihy

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661, and in part by the National Science Foundation under grant MCS 74-21892 A01

This blank page was inserted to preserve pagination.

Transmitting Abstract Values in Messages

Maurice Peter Herlihy

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661, and in part by the National Science Foundation under grant MCS 74-21892 A01.

Massachusetts Institute of Technology
Laboratory for Computer Science

Cambridge **Massachusetts 02139**

This empty page was substituted for a blank page in the original document.

Transmitting Abstract Values in Messages
by
Maurice Peter Herlihy

Abstract

This thesis develops primitives for a programming language intended for use in a distributed computer system where individual nodes may have different hardware or software configurations. Our primitives are presented as extensions to the CLU language. We assume that differences in hardware and in administrative policy require that individual nodes be free to choose their own local representations for common types, including user-defined types. Our main objective is to provide primitives to communicate values of user-defined type. Our primitives support a large degree of node autonomy, without requiring that communicating nodes have prior knowledge of one another's special characteristics. We argue that the precise meaning of value transmission is type-dependent; thus the user, not the language, must control the meaning of transmission for values of a type.

Thesis Supervisor: Barbara H. Liskov
Title: Associate Professor of Electrical Engineering and Computer Science

Keywords: Abstract Types, Distributed Systems, Message Passing, Modularity, Object-Oriented Programming, Programming Languages, Programming Methodology

This empty page was substituted for a blank page in the original document.

Acknowledgements

I owe special thanks to my advisor, Professor Barbara Liskov, for her editing, suggestions, criticism, and insights. I would also like to thank the members of the Distributed Systems Group, most especially Russell Atkinson, Toby Bloom, Paul Johnson, Eliot Moss, Eugene Stark, Craig Schaffert, and Robert Scheifler, who have all listened to me with great patience, and whose ideas and suggestions were indispensable. Finally, I would like to thank Ellen Laviana for her encouragement and support.

Submitted to the Department of Electrical Engineering and Computer Science on April 25, 1980 in partial fulfillment of the requirements for the Degree of Master of Science.

This empty page was substituted for a blank page in the original document.

CONTENTS

Introduction

Distributed computer systems have a greater potential for decentralized physical and administrative control than do more traditional centralized systems. It is felt that organizations consisting of co-operating, largely autonomous groups can best be served by computer systems consisting of collections of co-operating, autonomous *nodes*, where each node is controlled by a particular group [Reed 78, Svobod 79]. When we say that nodes are autonomous, we mean that the group controlling a node has a certain amount of freedom to choose its hardware configuration, and to run specialized or proprietary software. Nodes may perform specialized tasks, such as printing, or high-precision floating point arithmetic, and may benefit from specialized hardware configurations. Nodes owned by groups interested in special applications may be required to run private software. Rich groups may maintain expensive, sophisticated machines, while groups with smaller budgets may be limited to simpler devices.

Conflicting with the need for diversity and specialization is a need for individual nodes to co-operate and communicate. The existence of diversity in hardware, software, and administrative policy threatens to complicate the task of designing and verifying programs that involve the participation of several nodes.

A high-level programming language suitable for constructing distributed programs should support the specification of node behavior in a clear, verifiable, implementation-independent manner. Languages that support the use of data abstraction, such as CLU [Liskov 79], or Alphard [Wulf 76] already present a methodology for the construction of clean, modular interfaces between layers of a centralized system. To support communication and co-operation in a heterogeneous distributed system, it is desirable to impose interfaces with similar modularity qualities

between nodes.

This thesis develops communication primitives for a high-level language intended for writing distributed programs in a heterogeneous system. Communication among nodes is accomplished by message-passing, so that the behavior of a node can be completely characterized by the messages it sends and receives. Our primitives are structured to facilitate the design of distributed programs in terms of the message-passing behavior of participating nodes, independently of how the nodes implement that behavior.

We assume that communicating programs use the primitives developed in this thesis. Messages contain values such as integers, booleans, or values of user-defined type. We shall see that it is a relatively simple matter to communicate values of language-defined type; a node may send the integer value 1 to another node, even if the two nodes do not implement integers in the same way. In this thesis we address the more difficult problem of developing a well-structured language mechanism to communicate values of user-defined type.

1.1 Model of Computation

Following [Liskov 79a], the logical entities corresponding to individual administrative groups are called *guardians*. The physical machines on which guardians reside are called *nodes*. There is not necessarily a one-to-one correspondence between guardians and nodes, although guardians are abstractions of individual computers. A guardian has an address space containing *objects* and *processes*. A process is an execution of a sequential program; objects are CLU objects.

1.2 Model of Communication

With the exception of ports, to be discussed below, the address spaces of guardians are disjoint; guardians only communicate by message passing. Messages do not contain objects, they contain the *values* of objects. The result of sending a message containing an object's value is to create a new, distinct copy of that object at the destination guardian, having the same value as the original.

1.3 Language Primitives

The programming language used in this thesis is CLU [Liskov 79], with new primitives and data types to facilitate distributed programming. For simplicity, we ignore CLU's own variable facility, although we mention some of the issues it raises in the conclusion.

Port objects permit general routing and sorting of messages. Messages are addressed to ports, not guardians. Ports accept and store messages of pre-determined type, and they are the only objects that can be named across guardian boundaries.

The language includes **send** and **receive** primitives for communicating values of objects between guardians. Both **send** and **receive** specify a port. The **send** statement causes a message to be sent to the indicated port, and the **receive** statement causes a message previously received at the indicated port to be interpreted. A port object is created by a guardian, and only that guardian can process messages received by that port.

1.4 Multiple Representations

The CLU language provides a number of built-in data types, and permits users to define new types, which we call *abstract* types. Two kinds of information are useful for describing an abstract type T. *Specification* information describes the behavior of T objects in terms of a collection of primitive operations. *Representation* information includes the data structures used to represent T objects, and the code for the procedures implementing the primitive operations. Representation information is encapsulated within a *cluster.* Clusters are information hiding devices; other programs may use specification information about a type, but not representation information. This restriction is enforced by limiting access to an object's underlying representation to the primitive operations of the type.

Different guardians in the distributed system may implement the same abstract type. We do not require that all the guardians implementing a given type use the same representation. In fact, for many reasons it is desirable to allow different guardians to use different representations for a common abstract type. The most compelling reason is to realize the large degree of autonomy possible in a decentralized system. In a system of physically and administratively independent guardians, individuals will invariably be tempted to "customize" the implementations of common data types, while retaining the need to communicate their values with other guardians. For example, an individual may wish to install a privately developed hashing function in a guardian's implementation of a symbol table type.

Different patterns of use may encourage specialized representations; for example, a company's sales division may wish to support a more space-consuming representation of a telephone book, which, in addition to listing telephone numbers

and addresses keyed by names, lists numbers and names keyed by addresses, permitting more efficient canvassing of neighborhoods.

Hardware characteristics may also encourage specialized representations. A guardian whose underlying hardware interpreter directly supports complex arithmetic should treat complex numbers as a base-level type, and should not have to represent complex numbers in the same way as a guardian residing at a less powerful node. Similarly, guardians providing access to different kinds of photo-typesetting devices may use different internal representations for character fonts, while guardians that use those servers should use a single abstract *font* type, understood by all the servers, regardless of the underlying hardware interpreter.

Security concerns may also prompt a guardian to keep secret its representation for a type. The scheme developed in this thesis permits individual guardians to conceal the representation used to implement a type from other guardians implementing that same type.

1.5 Sharing

CLU objects may name other objects. When two objects name the same object, we say the latter is *shared.* The behavior of an object may depend, not only on the objects it contains, but also on sharing among them. The semantics of value transmission for such a type should state whether this sharing structure is preserved. Any scheme for transmitting values must address the problem of preserving (or not preserving) the sharing structure of objects. The scheme presented in this thesis takes the approach that the degree to which sharing is preserved is part of each type's definition. The language provides the implementors of a type with the tools necessary

to control the transmission of sharing structure.

1.6 Why Type-Independent Schemes Don't Work

A straightforward and general scheme for transmitting an object's value is simply to transmit the value of the object's underlying representation in terms of values of primitive type. Such a scheme clearly does not support multiple representations. Even if it were acceptable to force every guardian to use the same representation for each transmissible type, such a naive scheme would be completely unsuited for a language based on the use of data abstraction, as we discuss in the next paragraphs.

The underlying representation of an object may be transmissible, while the abstract value of that object may not be. For example, a file name may be represented by a character string. The string may be transmissible, but the file name may be meaningless outside of a particular file system belonging to a particular guardian.

Conversely, there are a number of situations where an object's abstract value is transmissible, but where the object's representation is unsuited as a vehicle for communicating its value. For instance:

> An object's representation may contain information meaningless to another guardian, such as an index into a private table maintained by the original guardian. A naive scheme could not recognize (and compensate for) such context-dependent information.
>
> An object's representation could include objects whose values are not themselves transmissible, (e.g. an I/O stream) but which can be reconstructed by the recipient.

> What constitutes the "value" of an abstract object may not always be clear from its representation. For example, each object of a type might be marked with its time of creation. When the value of such an object is transmitted, what creation time should the new copy contain? Only the programmer can make this decision.
>
> A type's representation may contain redundant information that may be more economically reconstructed than transmitted.

We conclude that transmissibility is a characteristic of an object's type, not of its underlying representation.

1.7 Related Work

We begin by providing a rather summary description of our scheme to lay a basis for comparison with previous work. We assume that the language implementations of the various guardians are capable of communicating values of built-in type. To communicate values of a user-defined type between guardians that may use different representations for that type, values are encoded into a standard intermediate representation, called the type's *external representation*. At the language level, this external representation takes the form of an object of different transmissible type. The external representation type may itself be user-defined, or contain user-defined types. When a value is sent in a message, a series of translation operations are invoked that eventually reduce the user-defined value to values of built-in type, which can be transmitted. Upon receipt, the inverse translations are applied to reconstruct the original value.

An alternative to standard intermediate representations is direct translation between representations. [Fabry 76] develops a scheme for replacing modules while the ambient system continues to run. During the transition from an old version to a

new version it is possible that different representations for objects of the same type may co-exist. In Fabry's scheme, each object is tagged with a version number, and each module version includes a translation operation from the representation used by the previous version to its own representation. Whenever an object using an old representation is encountered, a chain of translation operations is invoked to convert the object into the current representation for that type.

It does not appear that direct translation can be applied to the problem of value transmission in a heterogeneous distributed system. Fabry's version numbering scheme assumes that each new version makes a single predecessor obsolete, and thus it suffices to provide a single translation operation. In a heterogeneous system where each guardian may use a different representation, there is no such natural ordering among representations. When a new implementation of an existing type is introduced, how many translation operations must be provided? Must all other guardians be informed? How do guardians translate between hardware-dependent representations?

A number of schemes have emerged that permit transmission of built-in values between heterogeneous nodes through the use of standard intermediate representations [Levine 78,Crocker 75,Postel 74,White 74,Neigus 73,Telnet 73]. Our scheme builds on the results of these works, since we assume that the underlying language implementation can faithfully transmit such language-defined values as strings, or arrays of integers, independently of their machine-level representations.

[Levine 78] examines and evaluates different strategies for communicating values such as real numbers, integers, or files of characters among heterogeneous nodes. It is concluded that the use of standard intermediate representations best satisfies such criteria as flexibility, extensibility, and efficiency.

A number of protocols have been developed for transmission of typed information across the ARPANET.[1] The Procedure Call Protocol developed for the National Software Works [Crocker 75,Postel 74,White 74] is the most ambitious, being capable of transmitting such values as character strings, integers, and lists. The TELNET protocol [Telnet 73] is used for transferring character information, and the File Transfer Protocol [Neigus 73] is used to transfer files. In these protocols, the sender converts the information to be sent into a standard representation which is either statically determined, or agreed upon by negotiation. Upon receipt, the receiver converts the standard representation into whatever local representation it uses.

[Haber 78] discusses methods for dynamic replacement of modules managing collections of long-lived objects. Each module version includes operations to translate between its own representation and a "simple canonical" representation. When a new module encounters an object in the old representation, the old module version is called upon to translate the object into its canonical representation, and the new version translates the canonical representation into the current representation. It is remarked that canonical representations may be used to communicate values among heterogeneous nodes in a distributed system.

Our scheme differs from that described in [Haber 78] in that we explicitly state what constitutes a permissible external (canonical) representation. As we shall explain in detail in the next chapter, many of the modularity properties of our scheme are a direct result of the particular way external representations are defined.

---

1. By "typed" information, we mean as other than uninterpreted bit strings.

The PLITS language [Feldman 79] defines a number of language primitives for writing distributed programs. PLITS modules communicate by message-passing. Messages consist of individual values of unstructured primitive type.[1] The mechanisms used to communicate these values between heterogeneous nodes are not described. Users of the language who wish to transmit more complicated values such as arrays, or values of user-defined type, are left to their own devices.

When defining value transmission for a type, one must decide what constitutes the "boundary" of an object, and what effect transmission is to have on an object's sharing structure. A related problem, that of defining copying operations for objects in a distributed system is addressed in [Sollins 79]. The model of communication used in this thesis is similar to the *copy-full-local* operation described there. Our approach differs in that our primary interest is not in developing sophisticated copying operations; rather it is in developing language constructs to permit users to define transmissible abstract types in ways that do not compromise guardian autonomy.

[Gligor 79] discusses techniques for storing values of objects on secondary storage, using encryption to avoid compromising the security of the information. Their encryption scheme is largely independent of the message construction scheme developed in this thesis; it could be used to provide security and authentication to the language primitives developed here.

Both the choice of language primitives and the guardian model of computation used in this thesis have been taken from work done by the M.I.T. Distributed Systems

---

1. Integers, booleans, characters, and reals are suggested.

Project [Svobod 79, Liskov 79a].

1.8 Outline of the Thesis

The plan of this thesis is to present the value communication scheme at successively descending levels of abstraction. At the highest level, Chapter Two defines the communication primitives as extensions to CLU, and describes how the language user may define and implement transmissible abstract types.

Chapter Three outlines an implementation scheme for a run-time system supporting the language extension defined in Chapter Two. The mechanisms for constructing messages from objects and reconstructing objects from messages are spelled out in detail. To present the scheme as simply as possible, we postpone discussion of a number of efficiency-related issues.

Chapter Four addresses the issue of efficiency, describing optimizations to the implementation described in Chapter Three.

Chapter Five discusses the conclusions reached in the thesis, including the applicability of the methods developed here to other problem areas. Among these areas are: the storage of values on secondary memory, displaying values of abstract objects on terminals, and copying objects.

The Language Definition

This chapter describes a number of programming language primitives to support the communication of values among heterogeneous nodes in a distributed system. These primitives are presented as an extension to the CLU language. The extended language defines the meaning of transmission for built-in types, as well as providing the means to define and implement transmission for user-defined types. Some problems that arise when defining transmission for cyclic user-defined types are also addressed.

Rather than attempting to give a formal semantics for value transmission, this thesis presents informal definitions of the primitives introduced. A formal semantics for the extended language is a major undertaking in its own right, and lies beyond the scope of this thesis.

2.1 Goals of the Language

Before presenting the language design, we list a number of criteria that we feel any message-passing scheme should satisfy.

> The scheme should support multiple implementations of a single type without a combinatorial growth of complexity. In particular, the addition of new implementations of existing types must not require changes to existing implementations.
>
> The meaning of transmission for any given type should be determined by localized, single-level operations within the module implementing the type. Verification of these operations should suffice to verify the correctness of the module's implementation of value transmission.

> Message construction, transmission, and interpretation should be performed by the language implementation, not the user. The user should be able to indicate the objects whose values are to be transmitted, and the language implementation should do the rest.
>
> Any useful scheme must give the programmer a reasonably simple means to control the effect of transmission on sharing structure.
>
> Any useful scheme must be efficiently implementable. (However, we postpone discussing the efficiency of our scheme until a later chapter, after examining some possible implementations.)

2.2 Terminology

In subsequent discussions we adopt the following typographical conventions. Objects are denoted by letters in cursive script (*A*, *B*, *C*). Names of operations on objects are written in italics. We use CLU's dollar-sign notation to indicate the type associated with an operation, where applicable. For example, T$*similar* and T$*equal* are operations defined on T objects.

As in CLU, the basic containers for information are *objects*. The behavior of an object is determined by its *type*. Each type has an associated collection of operations to manipulate its objects. Objects have both an *identity* and a *value*. An object's identity determines which object it is, while its value is its information content. Objects of *mutable* type may change their associated values, while objects of *immutable* type may not. The identity of an object cannot change. Objects may *refer* to other objects. When an object refers to another, we sometimes say the former *contains* the latter. When two objects refer to the same object we say that the latter is *shared*. For a more complete description of CLU's model of computation, the reader is referred to the

CLU Reference Manual [Liskov 79].

We partition the types in CLU into three disjoint sets: primitive, abstract, and composite. *Primitive* types are unstructured, language-defined types such as **string**, **char**, **int**, **real**, and **bool**. *Abstract* types are user-defined types. *Composite* types are composed from language-defined type constructors, of which CLU has six: **array**, **oneof**, **record**, **sequence**, **struct**, and **variant**. Component types of a composite type may be either primitive, abstract, or composite. **Record**'s, **array**'s, and **variant**'s are mutable; the other composite types are immutable. Objects of composite type serve primarily to refer to collections of other objects. Primitive and composite types are sometimes referred to as *built-in* types. Primitive types and type constructors are required to be supported at every node, while an abstract type need only be supported at certain nodes.

A *cluster* encapsulates the implementation of an abstract type T by defining a *concrete representation* for T objects, and by defining T operations in terms of operations on T's concrete representation. The choice of concrete representation defines an *abstraction function* from values of the concrete representation type to values of the abstract type, denoted by T$*abstract*. There is no T$*abstract* operation available to users of the language.

In our discussion of transmission, it is useful to define precisely when we consider two objects to be *identical*, that is, when they have the same identity. The first requirement we make of any such definition is that only objects of the same type can be identical. Accordingly, we define T$*identical* to be an operation taking two T objects, returning **true** if and only if the arguments have the same identity. Note that *identical* is used only for explanatory purposes; there is no corresponding language

operation currently defined in CLU. T$*identical* is defined in the following way:

> If T is primitive, then T$*identical* is equivalent to T$*equal*, where the latter is defined by the CLU Reference Manual.
>
> If T is composite, then two objects are identical if they are the results of the same invocation of the T$*create* operation.
>
> If T is abstract, then two objects are identical if their concrete representations are identical.

The *identical* operation is not quite the same as the CLU *equal* operation. For the primitive types, and for the mutable composite types, *identical* and *equal* are indeed equivalent. When defining an abstract type T, the CLU Reference Manual suggests that proper usage of the T$*equal* operation requires that:

> the *equal* operation should be an equivalence relation satisfying the substitution property; i.e. if two objects are *equal*, than one can be substituted for the other without any detectable difference in behavior.[p.80]

For types having well-defined *equal* operations, it follows that if two objects are *identical*, then they are necessarily *equal*, although the converse may not be true.

Perhaps the most important distinction between *identical* and *equal* is that *identical* is defined for every type, and is never defined in terms of user-defined operations. If defined at all, the *equal* operations of abstract types are defined in terms of user-defined operations.[1] The *identical* operations for all types are defined by the language, independently of any user-defined operations.

---

1. The *equal* operations of immutable composite types may also invoke *equal* operations of user-defined component types.

In our subsequent examples, we use "***A*** ≡ ***B***" as an abbreviation for "T$*identical*(***A***, ***B***)", and "***A*** = ***B***" as an abbreviation for "T$*equal*(***A***, ***B***)".

2.3 Communication Primitives

We restrict discussion to messages consisting of the value of a single object (which may, of course, contain other objects). In Chapter Five we will discuss some more general kinds of messages, but we will see that they introduce no new difficulties.

Objects of type **port** are used to identify the recipient of a message. Ports are parameterized according to the type of value they receive, e.g., a port of type **port[int]** can only receive the values of integers. The names of ports may be sent in messages; however, only the node that created a port may receive messages sent to that port.

Users may cause the value of an object to be sent to a port by executing a **send** statement, indicating the object whose value is to be sent, and the port to which it is to be sent. A message may be received by executing a **receive** statement, specifying the port from which a message is to be taken, the variable to which the resulting object is to be assigned, and the amount of time the user is willing to wait for the message to arrive. The language implementation provides buffering of messages between the time they are sent and the time they arrive.

At the most summary level of description, the result of sending the value of a T object is to create a new T object, whose value bears some relation to that of the original. The meaning of transmission for T can thus be characterized by a *transmit* operation, mapping T objects to T objects.

For a primitive type P, P$*transmit* creates a new P object having the same value

as the original. For example: "abc" ≡ **string**$*transmit*("abc"), 1 ≡ **int**$*transmit*(1), *etc.* We note that like T$*identical* and T$*abstract*, no explicit T$*transmit* operation is directly available to users of the language. The T$*transmit* operation is a device that serves to explain the meaning of value transmission.

2.4 Transmitting Composite Types

The definitions given here concern only the values of objects; by discussing transmission in terms of values, rather than object identities, we sidestep the problem of defining the relation of sharing structure to value. This problem is addressed in a later section.

Transmission for a value of composite type is defined in terms of component transmission. For example, transmission for the **array**[T] type is defined informally as follows: the result of transmitting an **array**[T] is to create a new **array**[T] object, having the same bounds as the old array. Furthermore, the values of the new array's elements are the transmitted values of the old array's elements.

Transmission for the other composite types can be defined similarly. Let ***A*** be an object of composite type T. When the value of ***A*** is sent by a node, ***A***'s component objects are transmitted in some canonical order (e.g.: ascending order for **array**'s, lexicographical order for **record**'s). When the T value is received, the values of the components are received in canonical order, component objects are constructed, and a new T object is created and initialized from the component objects.

2.5 Transmitting Abstract Types

The definition of a transmissible abstract type specifies the meaning of transmission for that type by defining a *transmit* operation. Just as correct usage demands that the **copy** operation for an abstract type preserve the value of the object being copied, correct usage demands that the *transmit* operation for an abstract type preserve the value of the transmitted object. In other words, the information content of the received object should be the same, in some sense, as the information content of the sent object. The problem of defining *transmit* for an abstract type T is thus the problem of deciding which properties of T objects constitute their values, and what constitutes preservation of those properties. An important area where such issues arise is the question of the relation of value to sharing structure. Some of these issues are discussed in the section on sharing.

2.5.1 Implementing Transmissibility

We say that a type is *transmissible* if it has a *transmit* operation. For an abstract type T, the *transmit* operation is defined in the following way. A transmissible type XT is chosen, called the *external representation* type of T, along with a mapping from values of T to values of XT. This mapping is denoted by T$*encode*, and the inverse mapping by T$*decode*. The value of the object created by T$*transmit* is defined by the composition of T$*encode*, XT$*transmit*, and T$*decode*:

T$*transmit*(*A*) ≡ T$*decode*(XT$*transmit*(T$*encode*(*A*))).

The external representation of an abstract type T is specified by the definition of T; thus all clusters implementing T use the same external representation. The external representation type may be abstract, or composed from abstract types, but it must be

transmissible.

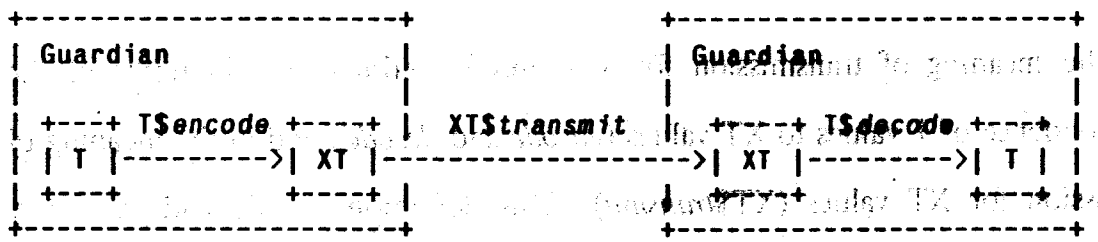
The meaning of transmission for T values is defined only in terms of the correspondence of T values to XT values (*encode* and *decode*), and in the meaning of transmission for XT values (XT$*transmit*). This definition is independent of any cluster's choice of concrete representation.

Each cluster implementing a transmissible type T must supply operations to implement the *encode* and *decode* mappings. The T$*encode* operation takes a T object, and returns an object of the corresponding external representation type, having the corresponding value. The T$*decode* operation performs the inverse mapping from an object of the external representation type to the corresponding abstract object. The *encode* and *decode* operations of a type are invoked automatically by the language implementation when a **send** or **receive** statement is executed.

A value of abstract type T is transmitted by the language implementation in the following way (Figure 1): When a node sends the value of a T object, T$*encode* is applied to the object, and the value of the resulting external representation object is sent (possibly by invoking further *encode*'s). When the target node receives the message, an external representation object is constructed from it, and T$*decode* is applied to it to produce an object of type T.

The *encode* and *decode* operations of a cluster encapsulate the translations between the concrete and external representations. These operations are completely defined within the cluster, contributing to modularity. To verify that a cluster correctly implements value transmission, it suffices to verify the cluster's *encode* and *decode* operations. The external representation also allows new T clusters to be written

Fig. 1. Definition of T$transmit

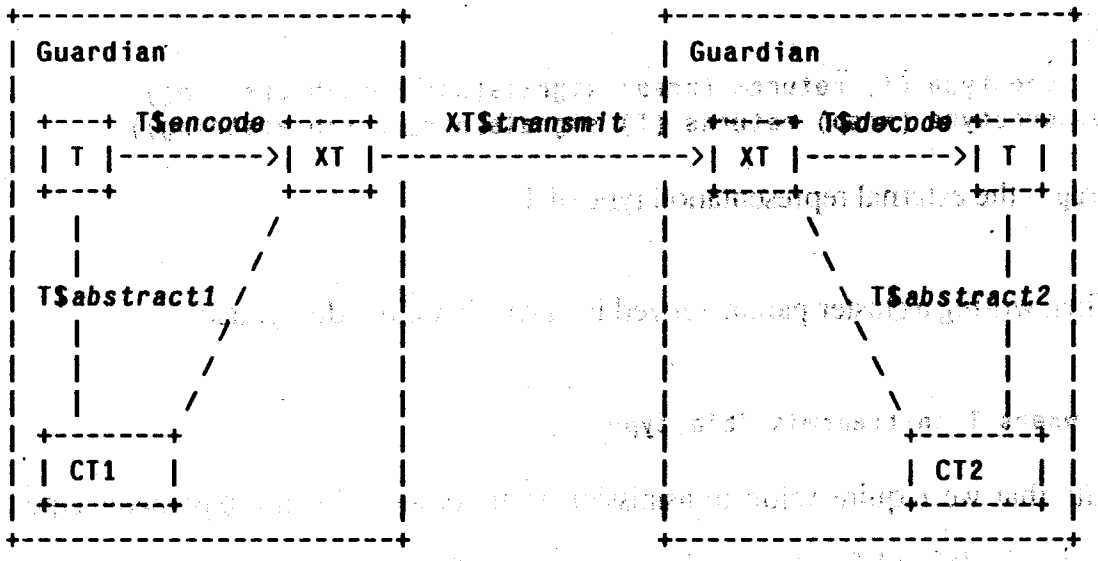


without affecting existing ones, since all T clusters communicate by converting local concrete representations for T values to XT values in a standard way as the values cross node boundaries.

Let T be a transmissible type supported at two guardians. Let CT1 and CT2 be the concrete representation types used by each, and XT the external representation type. As usual, let T$*encode* and T$*decode* denote the mappings between values of T and values of XT. Let T$*abstract*1 and T$*abstract*2 denote the mappings between values of CT1 and CT2 and values of T. The functionality of these mappings are illustrated in Figure 2.

The user of the T type needs to know the meaning of T$*transmit*, but he does not need to know the nature of T's external representation. The external representation of a type T is only of interest to the implementors of new T clusters. The meaning of transmission for primitive types and abstract types can be specified in the same way, without reference to whether values are transmitted directly or reduced to simpler transmissible values.

Fig. 2. The Relations of Encoding Operations



2.6 An Example

To serve as an example of a typical abstract type, we introduce a *single-key table* which stores pairs of objects, where one object (the *key*) is used to retrieve the other (the *item*). The single-key table type has operations for creating empty tables, inserting pairs, fetching the item paired with a given key, deleting pairs, and iterating through all key-item pairs.

To present the example, we define some simple syntactic constructs. As in CLU, the concrete representation for a type is declared within its cluster by use of the distinguished equate:

```
rep = type_spec
```

where "type_spec" stands for a type specification. In addition, the external representation for a type is declared by a similar distinguished equate:

```
xrep = type_spec
```

The interface specifications for the *encode* and *decode* operations of a transmissible type T are:

```
encode: proctype (T) returns (xrep) signals(not_possible(string))
decode: proctype (xrep) returns (T) signals(not_possible(string))
```

where xrep is the external representation type of T.

When writing a cluster parameterized by a type T we use the syntax:

```
where T in transmissible_types
```

to indicate that we require value transmission to be defined for the parameter type. Transmission is defined for the primitive types, and for abstract types having *encode* and *decode* operations. Transmission is also defined for composite types whose component types are transmissible.

Let us examine how a single-key table might be made transmissible. This type is of general utility, yet it admits many specialized concrete representations: a guardian that rarely deletes bindings might choose a representation that permits quick insertion and lookup operations, at the expense of the delete operation, while another guardian might use a proprietary hashing function, or a complicated list structure representation.

The most obvious candidate for this type's external representation is an array of key-item pairs. The *encode* operation for the single-key table creates an empty array of key-item pairs, extracts each pair from the table, and inserts it in the array. The *decode* operation creates an empty table, extracts each pair from the external representation object, and inserts it in the table. A sample implementation is shown in Figure 3.

Fig. 3. The Single-Key Table Type

```
table = cluster [key_type, item_type: type] is
        create,       % Create a new, empty table
        bind,         % Add a new key-item pair
        lookup,       % Given a key, return the associated item
        delete,       % Remove a key-item pair
        elements      % iterate through all key-item pairs

       where key_type, item_type in transmissible_types

    tab = table[key_type, item_type]
    pair = struct [key: key_type, item: item_type]
    rep = ... % complicated structure
    xrep = array[pair]

    .   % Code for other operations ...
    .
    .

    encode = proc(t: tab) returns (xrep)
        ans: xrep := xrep$new()
        for k: key_type, it: item_type in tab$elements (t) do
            xrep$addh( ans, pair${key: k, item: it})
            end % for
        return (ans)
        end encode

    decode = proc (x: xrep) returns (tab)
        t: tab := tab$create()
        for p: pair in xrep$elements(x) do
            tab$bind (t, p.key, p.item)
            end     % for
        return (t)
        end decode

    end table
```

2.7 Sharing

CLU objects may refer to other CLU objects. When an object is referred to more than once, we say that object is *shared.* Since mutable objects can be shared, the behavior of an object may depend not only on the values of its components, but on the

way those components are shared. Consider an **array** of objects of some mutable type T. If two elements of the **array** share a single T object, then a change to that object through one element will be observable as a change to the other. Alternatively, if the two elements contain distinct T objects, then a change to one element will not affect the other. Since the behavior of the two arrays is different, one can plausibly argue that they have different values, and that transmission should distinguish between them.

Although it may be useful to have the *transmit* operation for a type preserve sharing when transmitting a single value, it does not appear useful to preserve sharing between objects sent in distinct messages. To capture this aspect of transmission, we redefine the *transmit* operations to take a second argument: a *message context* that defines the scope of sharing preservation by identifying the message being transmitted. A message context can be viewed as uniquely identifying a particular execution of a **send** statement.

Let M be a message context, and let A and B be T objects. We further redefine the *transmit* operations to satisfy the following property.

T1: $A \equiv B \Rightarrow \text{T\$}transmit(A, M) \equiv \text{T\$}transmit(B, M)$

In other words, *transmit* preserves identity; if the value of an object is transmitted twice in the course of executing a single **send** statement, a single object is created at the receiving guardian.

Using the new definition of *transmit*, we will now specify the effect of value transmission on the sharing structure of objects of composite type, by informally stating a number of properties of **array[T]**$*transmit*. The notation used to present these properties is used for brevity. In the following statements, A and B denote any **array[T]**

objects, and *M* denotes any message context. The first two properties state that the resulting array has the same bounds as the original.

A1: **array[T]$low(array[T]$*transmit*(*A*, *M*)) = array[T]$low(*A*)**

A2: **array[T]$size(array[T]$*transmit*(*A*, *M*)) = array[T]$size(*A*)**

The third property states that the value of each component of the new array is the transmitted value of the corresponding component of the old array. Moreover, sharing of components is preserved.

A3: (∀ k) (k is a legal index of *A*) ⇒ T$*transmit*(*A*[k], *M*) ≡ array[T]$*transmit*(*A*, *M*)[k]

The *transmit* operations for the other composite types are defined similarly.

The language primitives developed in this thesis use the notion of object identity as the basis for defining the effect of the *transmit* operations on sharing structure. Object identity was by no means the only possible choice. One alternative, similar to that taken by the CLU *copy* operation for composite types, is to ignore sharing completely. The effect of applying the *copy* operation to an **array[T]**, *A*, (where T is a mutable type) is to create a new, disjoint **array[T]**, *A'*. Corresponding elements of *A* and *A'* have the same value, but any sharing among elements of *A* is not reflected by sharing among elements of *A'*.

A user wishing to preserve sharing in such a scheme must explicitly encode sharing information when the value is sent, and reconstruct it upon receipt. For example, to transmit an **array[T]**, *A*, preserving sharing among the elements, one might create an **array[T]**, *B*, referring to each T object in *A* only once, and an **array[int]**, *C*,

having the property that for every legal index k, *B*[*C*[k]] = *A*[k]. Sharing structure is explicitly encoded in the integer elements of *C*.

We reject this approach because we feel that sharing structure is part of an object's value, and so should be preserved by transmission. Moreover, although it is the responsibility of the language user to define and implement the effect of transmission on the sharing structure of an abstract type, the language definition should make the most common and useful definitions easy to implement. Just as for composite types, the sharing structure of an abstract type is part of its value. It is our opinion that a well-structured definition of value transmission for an abstract type should have properties analogous to A3; i.e., it will preserve its own internal sharing structure.

Another approach is to use *equal*, rather than *identical*, as the basis for preserving sharing. This approach has the drawback that the *equal* operation for abstract types are necessarily user-defined, while *identical* is not. To implement value transmission so as to preserve *equal*, each transmissible type would have to provide an *equal* operation, an awkward requirement. Moreover, a language implementation that must frequently invoke user-defined *equal* operations is likely to be much less efficient than one that can perform an implementation-defined check for object identity. In a recent implementation of this scheme by the author, testing for identity of composite objects is done by a simple test for pointer equality.

The descriptions of the transmission algorithms given in this chapter suffice to determine the order of application of *transmit* operations when a value is transmitted. Invocations of *encode* and *decode* operations caused by the application of *transmit* operations may be observable by the user, since *encode* and *decode* are user-defined,

and may have side-effects. Accordingly, we specify that for each object whose value is transmitted in a given message context, *encode* is invoked at the sending guardian at least once, and *decode* is invoked at the receiving guardian at least once. The language definition places no restrictions on the order or number of those invocations.

2.8 Two Examples

We use the table type to illustrate the two kinds of sharing properties that are of interest to the definer of a type's *transmit* operation. The first property concerns the effect of transmission on *internal* sharing structures. Suppose a single item I is bound to two keys K_1 and K_2 in a table T. Let the value of T be transmitted in a message context M, and let $T' \equiv$ table\$*transmit*(T, M). For any reasonable definition of table\$*transmit*, T' will contain keys K_1' and K_2' corresponding to K_1 and K_2 in T. By the effect of transmission on internal sharing we mean that the definition of table\$*transmit* should specify whether K_1' and K_2' continue to share a single item, or whether they are each bound to disjoint items.

The second important sharing property concerns the effect of transmission on sharing relations between *distinct* objects whose values are sent in the same message. By contrast to internal sharing, which concerns sharing relations within a single object, we call the second sharing property *external* sharing. Let T_1 and T_2 be tables sharing a single item I. Suppose the values of T_1 and T_2 are transmitted in a single message, where $T_1' \equiv$ table\$*transmit*(T_1, M), and $T_2' \equiv$ table\$*transmit*(T_2, M). By the effect of transmission on external sharing we mean that the definition of table\$*transmit* should specify whether T_1' and T_2' continue to share a single item, or whether they contain disjoint copies of I.

The only way we have provided for table$*transmit* to preserve external sharing of items is to have it invoke item$*transmit* on those items. Accordingly, we define the effect of transmission on the internal and external sharing structures of the table type in the following way. Let M be a message context, T a table, and K a key in the table.

TAB1: table$lookup(table$*transmit*(T, M), key$*transmit*($K$, M)) ≡
item$*transmit*(table$lookup(T, K), M))

TAB1 guarantees two properties. First of all, it guarantees that sharing of items within a given table is preserved. Secondly, it guarantees that sharing of items between distinct tables is preserved when the tables' values are transmitted in a single message context. Let M be a message context, and let T_1 and T_2 be (not necessarily distinct) single-key tables. Suppose:

$$I \equiv \text{table\$lookup}(T_1, K_1) \equiv \text{table\$lookup}(T_2, K_2).$$

Let:

$$T_1' \equiv \text{table\$}transmit(T_1, M)$$
$$T_2' \equiv \text{table\$}transmit(T_2, M)$$
$$K_1' \equiv \text{key\$}transmit(K_1, M)$$
$$K_2' \equiv \text{key\$}transmit(K_2, M)$$

By two applications of property TAB1:

$$\text{table\$lookup}(T_1', K_1') \equiv \text{item\$}transmit(I, M)$$
$$\text{table\$lookup}(T_2', K_2') \equiv \text{item\$}transmit(I, M).$$

By property T1, the right hand sides are identical, so sharing is preserved:

table$lookup($T_1'$, K_1') ≡ table$lookup(T_2', K_2').

An informal verification that the single-key table implementation listed above satisfies TAB1 is quite straightforward. By inspecting the code of the *encode* operation, we can see that if item I is bound to keys K_1 and K_2 in tables T_1 and T_2 respectively, then I is shared by two key-item pairs in the external representations, having keys K_1 and K_2. By a similar argument, *decode* also preserves sharing of items. To prove TAB1, it suffices to observe that **xrep$***transmit* preserves sharing of items, an observation that follows directly from the definition of *transmit* for the **array** types. If transmission of the external representation did not preserve sharing of items, then the *encode* and *decode* operations of the abstract type would have to be written to discover, encode and reconstruct the sharing structure.

If (for some perverse reason) the definition of the single-key table type had specified that transmission should *not* preserve external sharing of items, that effect could have been achieved by having the single-key table's external representation contain distinct copies of the item.

To illustrate how the *transmit* operation of a new type can be composed from the *transmit* operations of subsidiary types, let us examine the implementation and verification of a two-key table. A two-key table differs from a single key table in that it permits two types of keys to be used to retrieve items. A single item may be bound to any number of keys of either type. Just as for single-key tables, we require that if a single item is bound to several keys, then that sharing is preserved by transmission.

Since we already have a transmissible single-key table, let us choose, as an external representation for the two-key table, a **struct** consisting of two single-key

tables, each accepting one of the two types of keys.

```
xrep = struct[tab1: table1, tab2: table2]
```

We define the correspondence between values of the abstract type and values of the external representation type in the most straightforward manner: for each key-item pair in a two-key table, the appropriately typed single-key table component of the external representation contains the same pair. This definition implies that if the same item is paired with keys of different types in the two-key table, then that object will be shared by both single-key tables in the external representation.

We can verify informally that this choice of external representation preserves sharing of key-item pairs. From the definition of **xrep$***transmit*, we know that the value of each single-key table component is transmitted by its own *transmit* operation, using the same message context. Property TAB1 ensures that sharing of items both within a single-key table and between the two single-key tables is preserved. We observe that without property A3 to preserve sharing, we could not have constructed and verified the sharing properties of either table type as easily as we have.

To conclude the example, let us sketch an implementation for the two-key table. We choose a concrete representation identical to the external representation, with the same correspondence between concrete values and abstract values. The operations to add, delete, change, and retrieve pairs can be implemented in a straightforward manner. The *encode* and *decode* operations are particularly simple: they just return their argument after performing an **up** or a **down**. The implementation is illustrated in Figure 4.

Fig. 4. The Single-Key Table Type

```
two_key_table = cluster [k1_type, k2_type, item_type: type] is ...

   table1 = table [k1_type, item_type]
   table2 = table [k2_type, item_type]
   rep = record [tab1: table1, tab2: table2]
   xrep = rep

   .
   .
   .

   encode = proc (x: cvt) returns (xrep)
      return(x)
      end encode

   decode = proc (y: xrep) returns (cvt)
      return(y)
      end decode

   end two_key_table
```

2.9 Transmitting Cyclic Structures

When an object is created, CLU requires that it be given a value; there is no such thing as an uninitialized object in CLU.[1] This restriction adds to the safety of the language, since every object that can be named has a legal value.

Let *A* be an object of abstract type, and let *A'* be its external representation. We say that *A'* is *self-referential* if it refers to *A*. Values cannot be transmitted using self-referential external representations, as may be illustrated by the following example. Consider the *int_list* cluster shown in Figure 5 which implements linked lists of integers. The concrete representation is just a record with two components: the first

1. Although there may be uninitialized variables.

Fig. 5. Linked List of Integers

```
int_list = cluster is ...

   link = oneof [next: int_list, empty: null]
   rep = record[car: int, cdr: link]
   xrep = rep

   .
   .
   .

   encode = proc(x: cvt) returns(xrep)
      return(x)
      end encode

   decode = proc(y: xrep) returns (cvt)
      return(y)
      end decode

   end int_list
```

---

is an integer, and the second is either an int_list, or **null**. The external representation is the same as the concrete representation. We encounter a problem when we try to decode a message containing a circular list. To construct an int_list from a message, we must first have constructed its external representation. To construct the external representation object, we must first construct the objects it names. However, in the case of a circular int_list, the external representation contains the decoded int_list itself. The requirement that an object have a well-defined value before it can be named means that both the int_list and its external representation must be created before being named by the other, and thus neither can be constructed. Note that if the list is acyclic, then the external representation is not self-referential, and no such problem results.

It might appear reasonable to state that an external representation is not

well-formed if it is self-referential. Unfortunately, such a restriction makes the transmission of cyclic objects quite difficult. Consider the problem of making potentially cyclic int_list's transmissible. Whatever external representation we choose for the int_list type cannot itself contain an int_list component, since otherwise we cannot guarantee that the external representation is not self-referential. A simple strategy is to place the integer components in an array, along with some additional information indicating the index in the array of the element to which the last element was linked (with a special value for a **null** link). What the user is really doing here is evading the CLU requirement that every named object have a value, by disguising an object name as an array index.

We can take two approaches to the problem of transmitting cyclic structures. One option is to leave the implementors of cyclic types to their own devices when writing those types' *decode* operations. As a justification for this approach we might observe that language support for such transmission requires extending CLU's object semantics to permit naming objects before they are constructed, complicating both the language definition and its implementation.

The other option is to provide some explicit support for the transmission of cyclic structures. We have seen that either course forces the user to name objects before they have been given values. Without language support, the user must disguise the nature of such references from the language, a clear case of having the language hinder, rather than help the problem of program design. It seems unreasonable and inelegant to require the programmer to take heroic measures both to encode values and to circumvent the language definition.

Whether transmission of self-referential external representations is to be

supported is primarily a question of programming convenience, in the same way that implicit transmission of sharing information is a question of convenience. We have seen that without the ability to use self-referential external representations the transmission of cyclic structures becomes quite awkward. For this reason, we choose to relax the requirement that an object have a value before it can be named. However such references may only exist while a message is being decoded, and the implementation of the *decode* operation must satisfy certain restrictions.

The restriction we impose on *decode* operations can be informally summarized as follows: Let ***A*** be an object of abstract type T, and let ***A'*** be its external representation. ***A'*** may contain ***A*** if T$*decode* applied to ***A'*** does not use the *value* of ***A***. In other words, we allow ***A'*** to name ***A*** before ***A*** has been initialized, but we forbid T$*decode* to access the value of ***A***.

Let us make this notion more precise. Given a procedure P and an object ***A*** of abstract type T, we seek to formulate a rule that ensures that P does not depend on the value of ***A***. We do not require that this rule be exact, but we do require that it be conservative; whenever the rule is followed, we are safe, although we do not mind if the rule is overly strict.

Clearly, any procedure that operates on ***A***'s concrete representation uses its value. Moreover, the only procedures that can operate on ***A***'s concrete representation are the operations of the T cluster. This suggests the following rule: A procedure P *uses the value* of an object ***A*** of abstract type T if an invocation of P applies a primitive T operation to ***A***. This rule is safe, since without invoking an operation of the T cluster, P can only use the name of object ***A***. The rule is conservative, since it is possible that an operation of the T cluster might not access the concrete representation of ***A***.

If the *decode* operation constructing ***A*** uses the value of ***B*** by invoking a cluster operation on it, then the construction of ***B*** must precede the construction of ***A***. We say that ***A*** *depends on* the value of ***B*** if ***B*** is in the transitive closure of the "uses the value of" relation induced by applying *decode* to ***A***. If ***A*** depends on ***B***, then ***B*** must be decoded before ***A***. If ***A*** depends on itself, then its construction must precede itself, an obvious impossibility.

We may now make precise our restriction on *decode* operations that operate on self-referential external representations. A *decode* operation is legal if the "uses the value of" relation of the object being decoded is acyclic.

This restriction permits an object to be named by its own external representation, facilitating the transmission of cyclic structures. The int_list cluster as shown above will now legally transmit cyclic lists, since correctly decoding an int_list depends only on the identity of the following int_list, not on whether it has been initialized, as no operations are invoked on the successor.

We can display an illegal *decode* operation by choosing a different concrete representation for the int_list type (Figure 6). In this cluster's concrete representation, each element having a successor contains the value of the successor's integer, as well as its own. The external representation is the same as the one used above. Each int_list object depends on its successor, since *decode* invokes an int_list operation (car) on the next int_list. If the list is cyclic, an int_list depends on itself, and the *decode* operation fails the restriction. When implementing a new cluster for an existing transmissible type, it is the responsibility of the cluster writer to choose a concrete representation compatible with a legal *decode* operation.

Fig. 6. An Incorrect int_list Implementation

```
int_list = cluster is car, ...

    rep = record [car: int, cdr: link]
    link = oneof [non_empty: cdr_info, empty: null]
    cdr_info = record [next_list: int_list, next_car: int]

    xrep = record [car: int, cdr: xlink]
    xlink = oneof [non_empty: int_list, empty: null]

    car = proc(x: cvt) returns(int)
        return(x.car)
        end car

    encode = proc(x: cvt) returns(xrep)
        % Construct xrep's link
        xl: xlink
        tagcase x.cdr
           tag empty:
               xl := xlink$make_empty(nil)
           tag non_empty(ti: cdr_info):
               xl := xlink$make_non_empty(ti.next_list)
           end     % tag
        return(xrep${car: x.car, cdr: xl})
        end encode

    decode = proc(y: xrep) returns (cvt)
        % Extract record components
        lk: link
        tagcase y.cdr
           tag empty:
               lk := link$make_empty(nil)
           tag non_empty(list: int_list):
               lk := link$make_non_empty(
                         cdr_info${next_car: int_list$car(list),
                                   next_list: list})
           end         % tag
        return(rep${car: y.car, cdr: lk})
        end decode

    end int_list
```

Curiously, we can encode and send a self-referential external representation with no apparent difficulty. The nature of this asymmetry between sending and receiving

can be illuminated by observing that on the sending side, a self-referential external representation names the argument to a *past* invocation of *encode*, whereas at the receiving side, such an external representation names the result of a *future* invocation of *decode*.

An Implementation Design

This chapter presents an implementation design for the value transmission scheme described in the previous chapter. We describe run-time machinery to construct messages from objects, and to reconstruct objects from messages. The mechanisms introduced here are intended primarily as explanatory devices. As a consequence, we have made no attempt to optimize run-time performance or to minimize the number of constructs used. Although we feel that questions of efficiency are extremely important, we also feel that the structure of the implementation can best be conveyed by postponing a discussion of efficiency-related issues to the next chapter. By presenting the complete implementation design in two stages, we hope to distinguish fundamental aspects of the implementation from details intended to enhance performance.

Throughout this chapter, we refer to the construction of a message denoting a value as *encoding* the value, and to the interpretation of a message denoting a value as *decoding* the value.

3.1 Some Useful Data Abstractions

This section defines some data abstractions used to build the value encoding and decoding mechanisms. Operation definitions follow the terminology of the CLU Reference Manual: *arg1*, *arg2*, etc. refer to the operation's arguments. The interfaces of some of the data abstractions used differ slightly according to whether they are being used to encode or decode values. Where appropriate, we prefix the names of abstractions used to encode values with the letter "e", and those used to decode values with the letter "d".

3.1.1 Message Streams

A *message stream* is an abstraction of the communication medium, encapsulating specific characteristics of the medium that are irrelevant at the level of abstraction addressed here, such as the protocols used, or when messages are really sent. There are two kinds of message streams: encoding streams, which are used to send an object's value to a foreign port, and decoding streams, which are used to receive a value previously sent to a local port.

Information is transmitted in discrete units called *tokens*. When a value is sent, an encoding stream is created, and the value is placed in the stream as a sequence of tokens. A decoding message stream releases tokens in the same order they were placed into the original encoding stream. The external representation mechanism ensures that the sequence of tokens used to encode a value is independent of the concrete representation used by a guardian.

Encoding message streams are implemented by the *estream* type:

open: **proctype(port) returns(estream)**

Creates an encoding stream used to send tokens to the indicated foreign port.

insert: **proctype(estream, token)**

Inserts a token into an encoding stream.

current: **proctype(estream) returns(stream_addr)**

Returns the stream address of the next token. Stream addresses (see below) are used to refer to tokens already in the stream.

close: **proctype(estream)**

Indicates that the user does not intend to use the stream for further output.

Decoding streams are implemented by the *dstream* type:

```
open: proctype(port, timeout) returns(dstream) signals(timeout)
```

Creates a decoding stream for reading tokens previously sent to the port indicated by *arg1*. If the message is delayed due to node failure or communication failure, the timeout argument indicates how long the user is willing to wait. If the indicated amount of time elapses without a message, a timeout exception is signalled.

```
extract: proctype(dstream) returns(token) signals(timeout)
```

Removes and returns the next token from the stream. If the next token does not become available for the amount of time specified in the timeout argument to the *open* operation, a timeout exception is signalled and the stream is disabled.

```
peek = proctype(dstream) returns(token)
```

Behaves just like extract, except that it does not remove the next token from the stream. This operation will not be used until the next chapter.

```
current: proctype(dstream) returns(stream_addr)
```

Returns the stream address of the most recently extracted token.

```
close: proctype(dstream)
```

Indicates that the user does not intend to use the stream for further input.

3.1.2 Tokens

There are three kinds of tokens (Figure 7). *Header tokens* (Figure 8) mark the start of a new value of composite or abstract type. They may contain type or size information. *Back reference tokens* contain the stream address of a token previously placed into the stream. Sharing is indicated by back reference tokens. *Data tokens*

Fig. 7. Token Type Definition

```
token = oneof [data: data_token, % Primitive type
               header: header_token,      % Composite or abstract
               back_ref: stream_addr]     % Indicates sharing
```

Fig. 8. Header Token Type Definition

```
header_token = oneof [
                   abstract_hdr:   null, % abstract value
                   oneof_hdr:      int,  % tag value
                   variant_hdr:    int,  % tag value
                   array_hdr:      record [low, size: int],
                   seq_hdr:        int,  % size
                   record_hdr:     int,  % number of selectors
                   struct_hdr:     int   % number of selectors
                      ]
```

(Figure 9) represent values of primitive type such as integers, strings, booleans, etc. A *stream address* uniquely identifies a token in a given message stream.

Fig. 9. Definition of Data Token Type

```
data_token = oneof [
        bool:       bool,
        char:       char,
        int:        int,
        null:       null,
        real:       real,
        string:     string]
```

3.1.3 Maps

We recall from the previous chapter that if the value of the same object is sent twice in the same message, then a single corresponding object is constructed by the receiver. Objects of type *map* are used to ensure that transmission preserves sharing. A map contains corresponding pairs of objects and stream addresses. There are several kinds of maps. When encoding values, the *emap* type is used to locate the stream address of a given object's encoded value. When decoding values, the *dmap* type is used to locate the object constructed from the value encoded at a given stream address. The emap type has the following operations:

create: **proctype() returns(emap)**

Creates an empty encoding map.

enter: **proctype[T: type](emap, stream_addr, T) signals(exists)**

Enters *arg2* as the stream address where the encoded value of *arg3* starts. If *arg2* has already been entered, *exists* is signalled.

seen: **proctype[T: type](emap, T) returns(bool)**

If *arg2* has been entered, the result is **true**, otherwise the result is **false**.

lookup: **proctype[T: type](emap, T) returns(steam_addr) signals(not_found)**

If *arg2* has been entered, the associated stream address is returned, otherwise not_found is signalled.

The dmap type has the following operations:

create: **proctype() returns(dmap)**

Creates an empty decoding map.

enter: **proctype[T: type](dmap, T, stream_addr) signals(exists)**

Enters *arg2* as the object decoded from the value at the stream address denoted by *arg3*. If *arg2* has already been entered, *exists* is signalled.

```
lookup: proctype[T: type](dmap, stream_addr) returns(T)
    signals(not_found)
```

If *arg2* has been entered, the associated object is returned, otherwise not_found is signalled.

```
seen = proctype[T: type](dmap, stream_addr) returns(bool)
```

Returns **true** if an object of type T has been entered in the map with the given stream address. This operation will not be used until the next chapter.

Finally, we need a third kind of map that just remembers the identities of the objects that have been entered. We call this type the *initialization map*, for reasons that will be explained later. The initialization map is implemented by the *imap* type, and has the following operations:

```
create: proctype() returns(imap)
```

Creates an empty initialization map.

```
enter: proctype[T: type](imap, T) signals(exists)
```

Enters *arg2* in the map. If *arg2* has already been entered, *exists* is signalled.

```
elements: itertype[T: type](imap) yields(T)
```

Yields and removes all previous entries of type T.

```
empty: proctype(imap) returns(bool)
```

Returns **true** if there are no objects of any type currently in the map.

```
is_initialized: proctype[T: type](imap) returns(bool)
```

Returns **false** the first time it is invoked with the given parameter type, and

true thereafter.

3.1.4 Contexts

Objects of type *context* serve to associate the message stream and the maps used to encode or decode a single value. There are two kinds of contexts: encoding contexts and decoding contexts. The context types are defined by the following equates:

```
econtext = record [emap: emap, estream: estream]
dcontext = record [dmap: dmap, imap: imap, dstream: dstream]
```

3.2 The Algorithm for Encoding Values

The language implementation encodes an object's value by recursively traversing the object, rather like a LISP map function. As the object is traversed, the implementation creates tokens and places them in a message stream, using a map to keep track of sharing information.

Executing a **send** statement on an object of type T is equivalent to invoking the procedure shown in Figure 10. The **send** statement creates a new context for the message, opening a message stream and creating an empty map. T$*put* is then invoked to place the value of the T object in the stream as a sequence of tokens. After T$*put* returns, the stream is closed.

The language implementation provides every transmissible type with a *put* operation. The *put* operations are part of the language implementation; their existence is not visible to the user. The *put* operation for the type T has the following calling sequence:

Fig. 10. Effects of the Send Statement

```
send = proc[T: type](x: T, p: port[T])

    % Create a new encoding context.
    em: emap := emap$create()
    es: estream := estream$open(p)
    cxt: econtext := econtext${emap: em, estream: es}

    % Encode the value.
    T$put(x, cxt)

    % Close the stream.
    estream$close(es)

    end send
```

```
put: proctype(T, econtext)
```

The *put* operations for abstract and composite types preserve sharing in the following way. When *put* is invoked, it checks whether the object being sent has previously been entered in the map. If it has, then the associated stream address is extracted. A token containing a back reference to that stream address is put into the message stream, and T$*put* returns. If the object has not been previously encountered, it is entered in the map with its stream address. The *put* operation proceeds differently depending on whether its type is composite or abstract.

For an abstract type T, a header token is placed in the stream, and the external representation is created by applying T$*encode* to the T argument. XT$*put* is then invoked with the new external representation object and the old context. The *put* operation for an abstract T is illustrated in Figure 11.

For a composite type, a header token containing type-specific information is then placed in the message stream, and the *put* operations of the components are invoked.

Fig. 11. The Put Operation for an Abstract Type T

```
put = proc(x: T, cxt: econtext)

    % Has this object been seen before?
    if emap$seen[T](x, cxt.emap) then

       % Find the stream address of the object.
       back: stream_addr := emap$lookup[T](cxt.emap, x)

       % Output a back reference to the object.
       tok: token := token$make_back_ref(back)
       estream$insert(cxt.estream, tok)

       else
            % A new object, enter it in the map.
            next: stream_addr := estream$current(cxt.estream)
            emap$enter[T](cxt.emap, x, next)

            % Create and output a header token.
            htok: header_token := header_token$make_abstract(nil)
            tok: token := token$make_header_token(htok)
            estream$insert(cxt.estream, tok)

            % Create the external representation.
            y: xrep := T$encode(x)
            xrep$put(y, cxt)
       end % if

    end put
```

Figure 12 contains the text for **array[T]$*put*.**

If T is primitive, the object's value is encoded directly into a data token. Figure 13 contains the text for **int$*put*.**

Fig. 12. The Put Operation for the Array[T] Type

```
put = proc(x: array[T], cxt: econtext)

    % Has this array been seen before?
    if emap$seen[array[T]](x, cxt.emap) then

       % Find the stream address of the object.
       back: stream_addr := emap$lookup[array[T]](cxt.emap, x)

       % Output a back reference to the object.
       tok: token := token$make_back_ref(back)
       estream$insert(cxt.estream, tok)

       else
            % A new object, enter it in the map.
            next: stream_addr := estream$current(cxt.estream)
            emap$enter[array[T]](cxt.emap, x, next)

            % Create and output a header token.
            htok: header_token :=
                header_token$make_array(
                    array_hdr${low: array[T]$low(x),
                               size: array[T]$size(x)})
            tok: token := token$make_header_token(htok)
            estream$insert(cxt.estream, tok)

            % Output each element.
            for elm: T in array[T]$elements(x) do
                T$put(elm, cxt)
                end     % for
      end % if

    end put
```

Fig. 13. The Put Operation for the Int Type

```
put = proc(x: int, cxt: econtext)

   dtok: data_token := data_token$make_int(x)
   tok: token := token$make_data_token(dtok)
   estream$insert(cxt.estream, tok)

   end put
```

3.3 The Algorithm for Decoding Values

The language implementation decodes a value by removing tokens from the message stream and building up an object having the value represented. The implementation remembers the identities of objects constructed, so that when a back reference token is encountered, the system can identify the object indicated by the back reference.

The scheme described supports the use of self-referential external representations. To keep the explanation as simple as possible, we ignore the question of efficiency, and present a simple scheme that, in most cases, is more powerful than is strictly needed. In the next chapter we discuss ways of making this scheme more efficient.

3.3.1 Self-Referential External Representations

We recall that if ***A*** is an object, and ***A'*** its external representation, ***A'*** is *self-referential* if it contains ***A***. We stated in the previous chapter that to decode an object having a self-referential external representation it is necessary to name the object before it has been given a value. The implementation scheme adopted here permits an object to be named before its value has been reconstructed by creating a preliminary *uninitialized version* of the object. The identity of the uninitialized version is the identity of the object being decoded, although its value is undefined.

We emphasize that uninitialized versions are part of the language implementation, not part of the language. The user can never observe, or operate on, an uninitialized object version. Uninitialized versions can exist only while a **receive** is in progress.

3.3.2 Order of Initialization

Let *A* be an object of type T. When should an uninitialized version of *A* be used, and when should the completed version be used? As explained in the previous chapter, *A* cannot be initialized until the all the objects whose values are needed to initialize *A* have been initialized. This requirement has different implications for built-in types than for abstract types.

If T is primitive, then *A* is constructed from a single token. If T is composite, the value of *A* consists of the *identities* of its components, not their values. Thus, the initialization of an object of built-in type does not depend on any other object having been previously initialized.

If *A* is abstract, then *A* cannot be decoded until all the objects whose values are used by T$*decode* have been decoded. If any object whose *decode* precedes *A*'s refers to *A*, it must refer to an uninitialized version. In particular, no lower-level *decode* operation may invoke a T operation on an uninitialized version of *A*. Conversely, *A* must be decoded before any object whose *decode* operation depends on *A* can be decoded.

Reflecting the different degrees of dependency, values are decoded in two stages. In the first stage, called the *setup* stage, the values of primitive and composite type contained in the message are decoded. References to objects of abstract type are constructed as references to uninitialized object versions. No user-defined *decode* operations are invoked at this stage. In the second stage, called the *initialization* stage, all uninitialized object versions are initialized in a safe order.

Values of built-in type are decoded before values of abstract type because the

former can be efficiently decoded entirely by the language implementation. In particular, uninitialized versions of objects of composite type are protected from access by *decode* operations, since user-written procedures are not invoked until the initialization stage, by which time all objects of built-in type will have been initialized.

How do we prevent *decode* operations from operating on uninitialized versions of abstract objects? The order in which abstract objects must be decoded depends on the order in which their values are used by *decode* operations. Although this order might be determined by examining the text of all the *decode* operations invoked in the course of a receive, such an examination seems impractical. We choose to determine a proper order by initializing object versions only when an attempt is made to access the object's value. Since the values of the objects are constructed only when they are needed, this control structure is a kind of lazy evaluation [Friedman 76, Hender 76]. We call this strategy *lazy decoding*. When an operation of abstract type T is invoked from a *decode* operation, the language implementation checks each argument of type T to see whether it has been initialized. If it has, the invocation proceeds. If it has not, the current invocation is suspended, and the *decode* operation of the uninitialized object is invoked to initialize it. As soon as all T arguments have been initialized, the suspended invocation is resumed. This strategy guarantees that objects are decoded in an order consistent with the dependency relations described above, since an object is always decoded before its value is used, and no object is decoded prematurely.

Executing a **receive** statement is equivalent to invoking the procedure shown in Figure 14. The **receive** statement creates a new decoding context, opening a message stream and creating an empty map. T$*get* is then invoked to implement the setup stage, and T$*initialize* is invoked to implement the initialization stage. Finally, a

cleanup procedure is invoked to reclaim some unneeded storage.

3.3.3 Representation of Uninitialized Object Versions

We assume the language implementation uses object references of fixed size, [Snyder 79], as do all current CLU implementations. Use of fixed-size references means that it is possible to determine the storage required by an object from the information in its header token. In this way, we can allocate storage for an object of composite type before decoding that object. In this implementation, a reference to an uninitialized object version of composite type is a reference to the storage that will eventually be used by the initialized version.

We construct the uninitialized version of an object of abstract type by

---

Fig. 14. Effects of the Receive Statement

```
receive = proc[T: type](p: port[T], time: timeout)
        returns(T) signals(timeout)

    % The setup stage:
    % Create an empty decoding map.
    dm: dmap := dmap$create()
    % Open a message stream.
    ds: dstream := dstream$open(p, time)
    % Create an empty initialization map.
    im: imap := imap$create()
    cxt: dcontext := dcontext${dmap: dm, imap: im,
                               dstream: ds}
    x: T := T$get(cxt) resignal timeout
    dstream$close(ds)

    % The initialization stage:
    T$initialize(cxt)
    x := cleanup[T](x)
    return(x)

    end receive
```

constructing the object's external representation. The version is initialized by decoding the external representation. Unlike composite types, the uninitialized and initialized versions of an abstract object cannot use the same storage, since the latter is constructed from the former by a user-defined operation, and the language implementation has no way of knowing how large the result will be.

Since we cannot pre-allocate storage, every object of abstract type created during a **receive** is referred to indirectly through a *ufo* (unfinished future object). The representation of a *ufo* is shown in Figure 15. The meanings of the four states are as follows: The *ufo* is in the *empty* state when it is created. The *ufo* represents an uninitialized object version while it is in the *uninitialized* state, when it contains the object's external representation. When the *ufo* enters the *initialized* state, the object it represents has been initialized by decoding the external representation. The *in progress* state exists to detect illegal *decode* operations. A *ufo* is in this state while the object it represents is being constructed. If an attempt is made to access a *ufo* in this state, then a cyclic dependency exists and **failure** is signalled.

In addition, three operations are provided to detect and manipulate uninitialized object versions:

```
ufo_mask: proctype [T: type](ufo) returns(T)
```

Fig. 15. The Representation of a UFO

```
ufo = variant[empty: null,          % just created
              in_progress: null,    % being initialized
              uninitialized: any,   % xrep of represented object
              initialized: any      % represented object
              ]
```

Creates an uninitialized T object from the given *ufo*.

```
ufo_unmask: proctype [T: type](T) returns(ufo)
    signals(not_a_ufo)
```

If *arg1* is represented by a *ufo*, the underlying *ufo* is returned. Otherwise, not_a_ufo is signalled.

```
ufo_test: proctype [T: type](T) returns(bool)
```

If *arg1* is represented by a *ufo*, the result is **true**. Otherwise, it is **false**.

Lazy decoding is implemented in the following manner. Before the first line of any T cluster operation is executed, the language implementation tests each argument of type T to determine whether it is a *ufo*. If it is, an initialized version of the T object it represents is extracted, possibly by decoding the object's external representation. We call this test the *careful prologue* of an operation, and we assume it is automatically performed by the language implementation. A careful prologue is shown in Figure 16.

When receiving a value of type T, The setup stage is implemented by a T$*get* operation, which is provided to each transmissible type by the language implementation. Like the *put* operation, *get* is part of the language implementation, and is not visible to the user. The *get* operation for the type T has the following interface specification:

```
get: proctype(dcontext) returns(T) signals(timeout)
```

The *get* operation for an abstract type returns an *uninitialized version* of the object being decoded. The *get* operation for a composite type constructs the composite object (however, components of abstract type will refer to uninitialized versions). The *get* operation for a primitive type constructs the primitive object.

Fig. 16. The Careful Prologue of a T Cluster Operation

```
T = cluster is op, ...

    rep =  ...
    xrep =  ...
    .
    .
    .
    op = proc(arg: T)

        % Assign the initialized T object to variable "arg".

        if ufo_test[T](arg) then
           u: ufo := ufo_unmask[T](arg) % convert to ufo
           tagcase u
              tag empty, in_progress:
                  signal failure("illegal decode")

              tag initialized(a: any):
                  arg := force[T](a)

              tag uninitialized(a: any):
                  y: xrep := force[xrep](a)
                  ufo$change_in_progress(u, nil)
                  arg := T$decode(y)
                  ufo$change_initialized(u, arg)
              end % tag
           end % if

        % Now execute the user-written code.
        .
        .
        .
        end op

    end T
```

To construct a T object when T is primitive, the corresponding data token is removed from the stream and used to allocate and initialize storage for the object. Figure 17 contains the text for **int$*get***.

The *get* operations for abstract and composite types preserve sharing in the

Fig. 17. The Get Operation for the Int Type

```
get = proc(cxt: dcontext)
        returns(int)
        signals(timeout)

    % Create a token and output it.
    tok: token := dstream$extract(cxt.dstream)
       resignal timeout
    tagcase tok
       tag data(dtok: data_token):
           tagcase dtok
              tag int(ans: int): return(ans)
              others: signal failure("unexpected token type")
              end % tagcase
           ans: int := data_token$value_int(dtok)
       others: signal failure("unexpected token type")
       end % tagcase
    return(ans)

    end get
```

---

following way. When *get* is invoked, the next token in the stream is extracted. If the token is a back reference token, the object referred to is retrieved from the map, and *get* returns. If the token is a header token, *get* proceeds differently depending on whether the type is abstract or composite.

For a composite type, the header token is used to determine the amount of storage required. The necessary storage is allocated, and a reference to the uninitialized storage is entered in the map to catch cyclic references by components. An object created by a lower-level *get* may refer back to ***A*** through the map, but no attempt will be made to operate on ***A***, as no *decode* operations are invoked until after the setup stage has initialized all of ***A***'s component references. The text for **array[T]$*get*** is shown in Figure 18.

Fig. 18. The Get Operation for the Array[T] Type

```
get = proc(cxt: dcontext)
        returns(array[T])
        signals(timeout)

    array_hdr = record [low, size: int]

    % Examine the first token.
    tok: token := dstream$extract(cxt.dstream)
       resignal timeout

    tagcase tok
       tag back_ref(addr: stream_addr):
           % Object is old, look it up:
           return(dmap$lookup[array[T]](cxt.dmap, addr))

       tag header(hdr: header_token):
           % Object is new, allocate storage:
           ahdr: array_hdr := header_token$value_array_hdr(hdr)
           ans: array[T] := array[T]$predict(ahdr.low, ahdr.size)

           % Enter the object in the map.
           addr: stream_addr := dstream$current(cxt.dstream)
           dmap$enter[array[T]](cxt.dmap, addr, ans)

           % Get the components.
           for i: int in int$from_to(1, ahdr.size) do
               array[T]$addh(ans, T$get(cxt))
                  resignal timeout
               end
           return(ans)

       others: signal failure("unexpected token")
       end % tag

   end get
```

For an abstract type T, an empty *ufo* representing ***A*** is entered in the decoding map, bound to the stream address of ***A***'s header token. The uninitialized version of ***A*** is entered in the initialization map. XT$*get* is invoked, returning the external representation (which may itself contain uninitialized object versions). The external representation is placed in the *ufo* representing ***A***, and the uninitialized version is

returned. T$*get* is illustrated in Figure 19.

Fig. 19. The Get Operation for an Abstract Type

```
get = proc(cxt: dcontext)
        returns(T)
        signals(timeout)

    % Examine the first token.
    tok: token := dstream$extract(cxt.dstream)
       resignal timeout


    tagcase tok
       tag back_ref(addr: stream_addr):
           % Object is old, look it up:
           return(dmap$lookup[T](cxt.dmap, addr))

       tag header(hdr: header_token):
           % Object is new, create uninitialized version
           u: ufo := ufo$make_empty(nil)
           ans: T := ufo_mask[T](u)

           % Enter the object in the initialization map.
           imap$enter[T](cxt.imap, ans)

           % Enter the object in the decoding map.
           addr: stream_addr := dstream$current(cxt.dstream)
           dmap$enter[T](cxt.dmap, addr, ans)

           % Construct the external representation.
           y: xrep := xrep$get(cxt) resignal timeout
           ufo$change_uninitialized(u, y)
           return(ans)

       others: signal failure("unexpected token")
       end % tag

    end get
```

3.3.4 The Initialization Stage

At the end of the setup stage, no objects of abstract type have been initialized, but all objects of composite or primitive type contained in the message have been fully constructed. In the initialization stage, all uninitialized object versions previously placed in the decoding context's initialization map are initialized.

The initialization stage can be viewed as an optimization, since it is not necessary for correctness to initialize all objects. The lazy decoding scheme guarantees that the values of abstract objects will be available when needed. Nevertheless, since *decode* operations may contain errors or cause side-effects, it is convenient to assure the user that all *decode* invocations have completed when the receive statement completes.

Each transmissible type T is provided with an *initialize* operation. Like *put* and *get*, *initialize* operations are provided by the language implementation, and may not be invoked by users. *Initialize* operations have the following calling sequence:

```
initialize: proctype(dcontext)
```

T$*initialize* iterates through the uninitialized object versions of type T that had previously been entered in the decoding context's initialization map, as well as invoking the *initialize* operations of subsidiary types. The *is_initialized* operation of the imap type prevents infinite recursion by detecting the second and subsequent attempts to initialize objects of a given type.

The *initialize* operation for a composite type T simply invokes the *initialize* operations of its subsidiary types. The text for array[T]$*initialize* is shown in Figure 20.

The *initialize* operation for an abstract type T iterates through the T objects

Fig. 20. The Initialize Operation for Array[T]

```
initialize = proc(cxt: dcontext)

    % Check that the invocation is new.
    if imap$is_initialized[array[T]](cxt.imap)
       then return end

    % Initialize the subsidiary type.
    T$initialize(cxt)

    end initialize
```

---

entered in the initialization map, extracts the *ufo*'s, and initializes them if they are uninitialized. When a *ufo* representing a T object is initialized, T$*decode* is invoked. Lazy decoding may cause other object versions to be initialized. T$*initialize* is shown in Figure 21.

The *initialize* operation for a primitive type returns immediately.

3.3.5 Cleaning Up

At the end of the initialization stage, initialized *ufo*'s remain in the representation of the object received. Since we assume (for now) that every abstract operation has a careful prologue, it is not necessary for correctness to remove *ufo*'s. Removing *ufo*'s does improve the performance of abstract operations, so it may make sense to remove them at the end of the initialization stage. For this purpose, we use a *cleanup* operation:

```
cleanup = proctype[T: type](T) returns(T)
```

Cleanup performs a mark-and-sweep traversal of the machine-level representation of

Fig. 21. The Initialize Operation for an Abstract Type T

```
initialize = proc(cxt: dcontext)

    % Check that the invocation is new.
    if imap$is_initialized[T](cxt.imap) then return end

    % Initialize subsidiary types.
    xrep$initialize(cxt)

    % Initialize subsidiary objects.
    for x: T in imap$elements[T](cxt.imap) do
        u: ufo := unmask_ufo[T](x)
        tagcase u
           tag initialized: % nothing to do

           tag uninitialized(a: any):
               % Extract external rep object.
               y: xrep := force[xrep](a)
               ufo$change_in_progress(u, nil)
               x := T$decode(y)
               ufo$change_initialized(u, x)

           others: signal failure("illegal decode")
           end % tag
        end % for

    end initialize
```

---

its argument, replacing references to initialized *ufo*'s by direct references to the contained objects.

3.4 An Example

To illustrate how these mechanisms work, we trace how the value of an object consisting of two simple (and rather useless) mutually recursive types is transmitted. An *engine* object has a serial number and an optional caboose. A *caboose* object has a color and an associated engine.

The external representation of an engine is a **record** having as components a serial number of type **int**, and a **oneof** which is either **null** or contains a caboose. The cluster we examine here (Figure 22) uses the same concrete and external representations.

The external representation of a caboose is a **struct**, having as components a string denoting the color, and an engine. The concrete representation used by the cluster we examine also contains its engine's serial number in a *cache* component (Figure 23).

We observe that no operations of abstract type are invoked from the *decode*

Fig. 22. The Engine Cluster

```
engine = cluster is create, get_serial, ...

    train = oneof[empty: null, car: caboose]
    rep = record[rear: train, serial: int]
    xrep = rep

    .
    .
    .


    get_serial = proc(x: cvt) returns(int)
        return(x.serial)
        end get_serial

    encode = proc(x: cvt) returns(xrep)
        return(x)
        end encode

    decode = proc(x: T) returns(cvt)
        return(x)
        end decode

    end engine
```

Fig. 23. The Caboose Cluster

```
caboose = cluster is create, ...

    rep = struct[color: string, front: engine, cache: int]
    xrep = struct[color: string, front: engine]

    .
    .
    .

    encode = proc(x: cvt) returns(xrep)
        return(xrep${color: x.color, front: x.front})
        end encode

    decode = proc(y: xrep) returns(cvt)
        cache_val: int := engine$get_serial(y.front)
        return(rep${color: y.color,
                    front: y.front,
                    cache: cache_val})
        end decode

    end caboose
```

---

operation of the engine type. The caboose cluster's *decode* operation invokes an engine operation; thus, the caboose *decode* operation uses the value of the associated engine. From these observations, we conclude that the *decode*'s listed above are legal, since the transitive closure of the "uses the value of" relation is acyclic. When decoding a linked engine and caboose, the engine must be decoded before the caboose, since its value is used to initialize the caboose.

Let e be a variable bound to an engine, having serial number 9, and linked to a red caboose. Let p be a port[engine]. We will trace the effects of executing

```
send e to p.
```

First, a new encoding context is initialized. Then a number of *put* operations are

invoked. For brevity, when naming operations of composite type we use names like "**record**$*put*" when the particular **record** type is clear from context. Each invocation is listed with its depth from the top of the calling stack.

Level 1: engine$*put* checks the map to determine whether the engine has already been encoded. The engine has not been entered in the map, so *put* enters it, and places a header token in the stream at address 0. Engine$*encode* is invoked to (trivially) construct the external representation, then **record**$*put* is invoked on the result.

Level 2: Since **record**$*put* does not find its argument in the map, a header token is output at stream address 1 to indicate that the value of a **record** containing two selectors is starting. The **record** object is entered in the map, and the *put* operation of the first component (in lexicographical order) is invoked.

Level 3: After unsuccessfully checking the map, **oneof**$*put* outputs a header token at address 2 to indicate that the value of a **oneof** with a tag value of 1 is starting. *Put* is invoked on the caboose component.

Level 4: After unsuccessfully checking the map, caboose$*put* outputs a header token at address 3. caboose$encode constructs the external representation, then **struct**$*put* is invoked on the result.

Level 5: After unsuccessfully checking the map, **struct**$*put* outputs a **struct** header token with two selectors at stream address 4, then proceeds to invoke *put* on its first component.

Level 6: **string**$*put* outputs a token denoting the string value "red" at stream address 5.

Level 5: **struct**$*put* invokes *put* on its second (engine) component.

Level 6: engine$*put* finds the engine in the map, with associated stream address 0. It outputs a back reference to stream address 0 at stream address 6, and returns. Each of the suspended *put* operations at levels 5,4, and 3 also return.

Level 2: **record**$*put* resumes and invokes **int**$*put* on its second (serial) component, which places a token denoting the integer value 9 at stream address 7. All the suspended *put* operations then return.

When the highest-level invocation of engine$*put* returns, the stream is closed, and the **send** statement terminates.

To complete the example, we trace the effects of executing:

```
receive e on p.
```

First a decoding context is initialized. In the setup stage, the values of built-in type are constructed.

Fig. 24. Tokens Produced by Sending Engine Value

Stream Address	Token Type	Token Information
0	header	abstract value
1	header	record with two selectors
2	header	oneof with tag value 1
3	header	abstract value
4	header	struct with two selectors
5	data	string value "red"
6	back reference	stream address 0
7	data	int value 9

Level 1: engine$*get* extracts the first token, and determines that it is a header token. An empty *ufo* is entered in the encoding map, bound to stream address 0, and in the initialization map. **record**$*get* is invoked to construct the external representation.

Level 2: **record**$*get* extracts the next token, and determines that it is a header token. Storage for a **record** having two selectors is allocated. The uninitialized **record** is entered in the map, bound to stream address 1, and the *get* operation of the first component is invoked.

Level 3: **oneof**$*get* extracts the next token, and determines that it is a header token with tag value 1. Storage for a **oneof** is allocated. The uninitialized **oneof** is entered in the map, bound to stream address 2, and *get* is invoked on the component.

Level 4: caboose$*get* extracts the next token, and determines that it is a header token. An empty *ufo* is entered in the decoding map, bound to stream address 3, and in the initialization map. **struct**$*get* is then invoked to construct the external representation.

Level 5: **struct**$*get* extracts the next token, and determines that it is a header token. Storage for a **struct** having two selectors is allocated. The uninitialized **struct** is entered in the map, bound to stream address 4, and the *get* operation of the first component is invoked.

Level 6: **string**$*get* constructs a string having the value "red" from the token at stream address 5.

Level 5: **struct**$*get* resumes, stores the value "red" in its first component, and invokes *get* on its second component.

Level 6: engine$*get* extracts the next token, and determines that it is a back reference to stream address 0. The empty *ufo* representing the engine is extracted from the map, and returned.

Level 5: after storing the *ufo* in its second component, **struct$***get* returns.

Level 4: caboose$*get* resumes execution. It changes its *ufo* to the uninitialized state by binding it to the external representation returned from the lower level. This uninitialized object version is returned.

Level 3: after storing its component *ufo*, **oneof$***get* returns.

Level 2: **record$***get* resumes execution, storing the **oneof** in its first component, and invoking *get* on its second component.

Level 3: **int$***get* constructs an **int** having the value 9 from the token at stream address 5.

Level 2: **record$***get* stores the value 9 in its second component, and returns. Engine$*get* then returns.

All the values of primitive and composite type sent in the message have been constructed. The result of the setup stage is shown schematically in Figure 25.

In the initialization stage all the *ufo*'s created in the setup stage are initialized.

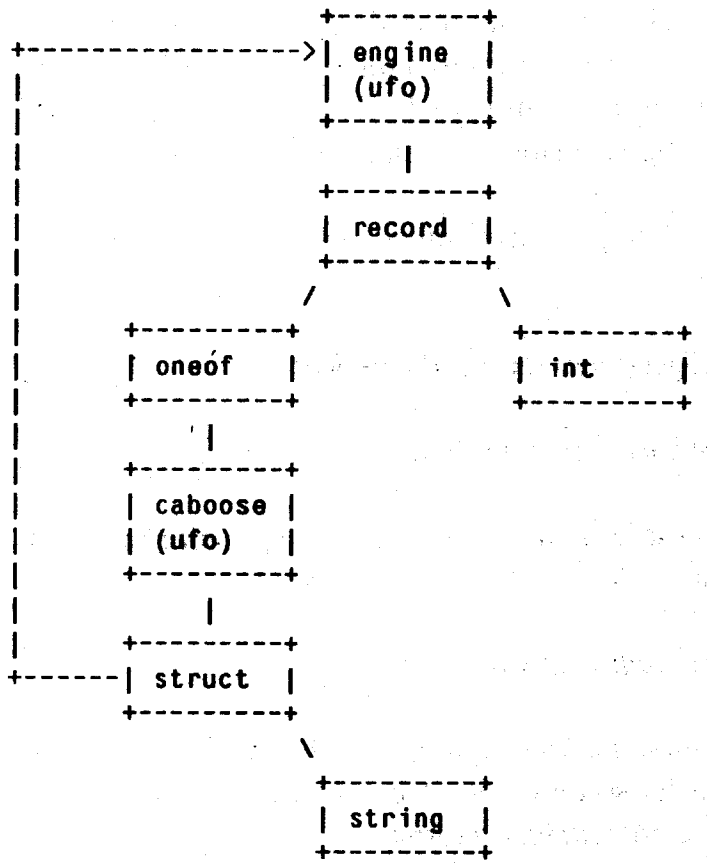
Level 1: engine$*initialize* invokes **record$***initialize*.

Level 2: **record$***initialize* invokes the initialize operation of its first component.

Level 3: **oneof$***initialize* invokes the *initialize* operations of its component types. **null$***initialize* returns immediately. Caboose$*initialize* is then invoked.

Level 4: caboose$*initialize* invokes **struct$***initialize*.

Fig. 25. The Results of the Setup Stage



Level 5: **struct$*initialize*** invokes ***initialize*** on its first (color) component. **string$*initialize*** returns immediately. Engine**$*initialize*** is then invoked.

Level 6: when engine**$*initialize*** invokes imap$is_initialized, it returns **true**, so engine**$*initialize*** returns.

Level 5: **struct$*get*** returns.

Level 4: caboose$*initialize* resumes execution and invokes imap$elements[caboose], which yields the *ufo* created at level 4 during the setup stage. The *ufo* is found to be uninitialized, so the external representation is extracted, and caboose$*decode* is applied to it.

Level 5: caboose$*decode* invokes engine$get_serial.

Level 6: the careful prologue of engine$get_serial determines that the engine argument refers to a *ufo*. The state of the *ufo* is tested and found to be uninitialized. Engine$*decode* is invoked to initialize the engine. (This is an example of lazy decoding.)

Level 7: engine$decode returns without invoking any operations of abstract type.

Level 6: engine$get_serial returns the integer 9.

Level 5: caboose$*decode* resumes, returning a caboose.

Level 4: caboose$*initialize* returns, having initialized all uninitialized cabooses.

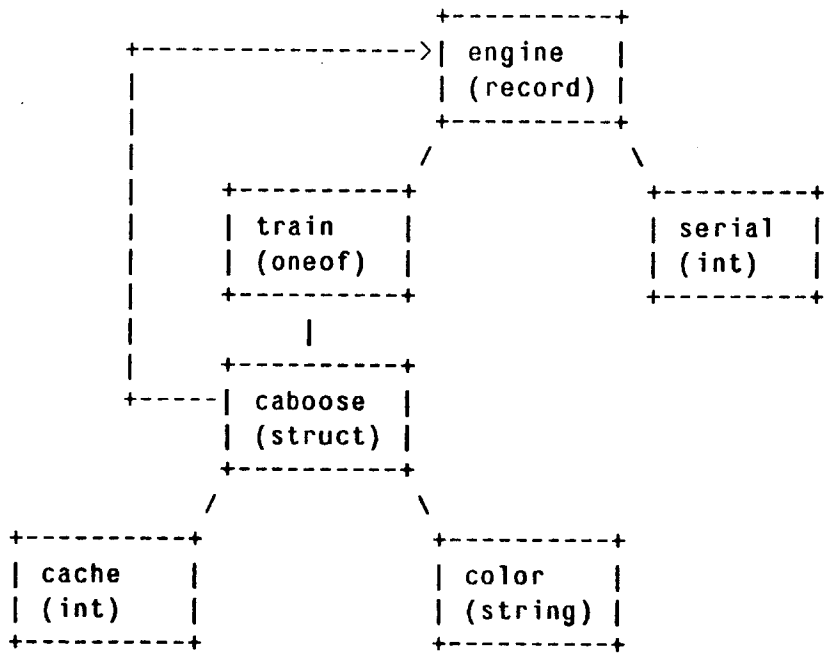
Level 3: **oneof$***initialize* returns.

Level 2: **record$***initialize* resumes execution. It invokes **int$***initialize* on its second (serial) component, which returns immediately. **record$***initialize* returns.

Level 1: engine$*initialize* resumes execution and invokes imap$elements[engine], which yields the *ufo* created at level 1 during the setup stage. The *ufo* is found to have been initialized (at level 6 above), so engine$*initialize* returns.

After engine$*initialize* returns, *cleanup* traverses the object and removes the initialized *ufo*'s from the representation. The result of receiving the message is to construct a linked engine and caboose, having the same values as the originals. The result is shown schematically in Figure 26.

Fig. 26. The Results of the Initialization Stage



Refinements and Optimizations

This chapter describes refinements to the implementation design presented in the previous chapter. We identify a number of common situations that do not require the full generality of the mechanisms we have introduced. For each situation, we explain how to recognize it when it occurs, and how to take advantage of it.

4.1 Overview

To help describe these refinements, we divide value transmission into two parts: value translation and message construction. Value translation is the task of translating between values of abstract type and values of built-in type. To transmit an abstract value, it is necessary to encode it into values of built-in type, since the lowest-level language implementation can only transmit built-in values. When sending a message, abstract values are reduced to built-in values through successive application of *encode* operations. When receiving a message, abstract values are constructed from built-in values through successive application of *decode* operations. In this chapter we address how to optimize the translation task.

The second task comprises the construction and transmission of messages containing values of built-in type. The mechanisms described in the previous chapter are designed to transmit values in a way that assumes as little as possible about the implementations of built-in types used at the communicating guardians. In an actual implementation, we may expect that some common patterns of communication will not require the full generality of the mechanisms we have described. In particular, when the sender and receiver reside on the same machine we may take advantage of the fact that both sides of the exchange may share memory, and may use the same

implementations of built-in types.

Two preliminary definitions are in order: a *module* is the unit of compilation, and *binding* is the process of combining separately compiled modules to form a program.

4.2 Translating Between Abstract and Built-in Values

The greatest apparent threat to efficiency in the translation task is the lazy decoding mechanism introduced in the previous chapter. We recall that lazy decoding requires that each operation of abstract type execute a careful prologue to test whether certain arguments are uninitialized. Although we can make testing itself quite efficient, we would like to reduce its frequency.

There are two complementary approaches to reducing the expense associated with lazy decoding. The first approach is to distinguish between those operation invocations that may encounter uninitialized object versions, and those that cannot. Uninitialized object versions can exist only while a message is being decoded. If we make the plausible assumption that most invocations of cluster operations occur while a **receive** is not in progress, then it becomes attractive to distinguish between invocations that may need to perform careful prologues, and those that do not. We present a scheme that restricts the execution of careful prologues to invocations of cluster operations that occur in the course of message decoding.

A second approach is to identify data types whose implementations do not need lazy decoding. For example, we recall that lazy decoding was introduced to permit the use of self-referential external representations to transmit values of cyclic objects. Realistically, we expect that only a minority of types include cyclic objects, suggesting that methods for statically recognizing types that only include acyclic objects may be

profitable. We present two schemes for recognizing that a given cluster does not require lazy decoding.

4.2.1 Restricting the Use of Careful Prologues

In this section, we discuss how to structure cluster operations to execute careful prologues only while a **receive** is in progress. We recall that uninitialized object versions are implemented by adding a level of indirection to object references. This level of indirection goes through an object we have called a *ufo*. When an operation of type T is invoked, it must check whether any of its T arguments is referred to indirectly, and if so, it must extract a direct reference from the intermediate *ufo*. We can increase the overall efficiency of abstract operations by ensuring that indirect references can exist only while a **receive** is in progress, and by executing careful prologues only at that time.

We divide modules into two classes: *careful* modules, which are prepared to encounter *ufo*'s in object representations, and *normal* modules, which are not. Only careful modules are allowed to execute when decoding a value. After the value is fully decoded, all indirect references through *ufo*'s are replaced by direct references. By having the binder create two versions of those modules that can be invoked both when a **receive** is in progress, and when one is not, we may avoid the expense of executing unneeded careful prologues, at the expense of the storage required for the extra module version.

The *cleanup* operation, previously introduced as an optimization, is necessary to ensure the safety of this scheme. Since normal modules do not expect to encounter indirect references, all *ufo*'s must be removed from the constructed object before any

normal modules resume execution.

The compiler only needs to produce one version of the object code for a module; differentiation of normal and careful versions may be done by the binder. We assume that the binder is aware of the interface specifications of cluster operations through the Library. In particular, the binder can determine which arguments to each T operation are T objects. If, by binding time, it has been decided that lazy decoding is necessary for a given T cluster, the binder can "enclose" the careful versions of cluster operations with dummy procedures that test and conditionally initialize T arguments before invoking the actual operation.

When joining modules, the binder follows these rules:

> Careful modules are bound to careful modules, and normal modules are bound to normal modules.
>
> All *decode* operations are careful.

When the same procedure is invoked from both careful and normal modules, the binder makes two copies of the procedure, placing a careful prologue in the careful version, if required.

In summary, we have shown how to limit the run-time penalty for lazy decoding to invocations that occur while a value is being decoded, at the cost of using more storage. This scheme requires only simple changes to the binder, which must distinguish between careful and normal modules.

4.2.2 Information About Abstractions and Implementations

In the remainder of this section, we discuss ways to detect that the objects managed by a given cluster can be decoded without creating uninitialized object versions. Our basic strategy is to collect information both about data abstractions and the modules implementing those abstractions, in order to establish that sufficient conditions exist to eliminate lazy decoding. We remove the need for careful prologues in cluster operations by substituting different *put*, *get*, and *initialize* operations from those described in the previous chapter.

There are two kinds of information that will prove useful. *Specification* information about a module concerns the abstraction it implements. Specification information includes such items as the names and argument types of procedures, and the external representation used by a data type. *Implementation* information concerns the way that a module implements an abstraction. Implementation information includes details such as a cluster's concrete representation, or the source code for a procedure.

We may also classify information by the ways it can be acquired. *Compile-time* information about a module is information that can be collected during or after the compilation of the module. Such information can be derived from implementation information about the particular module being compiled, along with specification information about the modules it uses. *Binding-time* information concerns implementation information about more than one module. Such information cannot be acquired until it is known which implementations are being bound together.

Information about modules and abstractions is managed by the *Library*. The

CLU Library [Liskov 79] contains the *interface specifications* of abstractions needed to type-check inter-module references. The Library for a CLU extension incorporating the communication primitives developed here would contain the external representations of transmissible data abstractions, since the external representation is the interface between distinct clusters implementing the same abstraction. The Library also maintains information about individual implementations. We assume that both the compiler and the binder can access and update information in the Library.

4.2.3 Eliminating Abstract Value Headers

In this section we show how to lower the number of tokens transmitted, at the cost of slightly complicating the control structure of the *get* operations. In itself, this reduction is not very important; however it permits us to optimize the case, discussed below, where the *encode* and *decode* operations of a type perform no actual work.

In the implementation presented in the previous chapter, the start of an encoded abstract value within a message stream is marked by a header token. In fact, the information conveyed by this kind of header token is redundant, since the type of a message is known in advance from the type of the port.

The optimized *get* operation acts in the following way. Let T be an abstract type having external representation type XT. When T$*put* encounters an object that has not previously been encoded, it invokes XT$*put* without placing a header token in the stream.

At the receiving guardian, care must be taken when a back reference token is encountered, since an encoded abstract value now starts at the same stream address as the encoded value of its external representation, and it is necessary to determine which

value is indicated. Accordingly, when T$*get* is invoked, it examines the next token in the stream without removing it. If the token is not a back reference, XT$*get* is invoked. If it is a back reference, the decoding map is checked to ascertain whether a T object has been entered with the given stream address. If such an object is found, the token is removed from the stream, and the object is extracted from the map and returned. If no associated T object is found, then one must be constructed, so T$*get* invokes XT$*get*, leaving the back reference token in the stream. T$*get* is shown in Figure 27.

To illustrate how this scheme differs from the previous one, let us compare how the two schemes would transmit a given value. Let ***A*** be an object of abstract type T, having the same object ***R*** as concrete and external representation. For the purposes of this example, let ***R*** be an **array[int]** with a single element. We suppose that ***A*** has an "exposed representation", that is, its concrete representation may be accessed and manipulated by programs other than T cluster operations. Let us transmit the value of a **struct** containing both ***A*** and ***R***.

The tokens produced by the unoptimized scheme appear in Figure 28. At the sending guardian, **struct**$*put* outputs a header token at stream address 0, and invokes **array[int]**$*put*, which outputs the tokens at stream addresses 1 and 2. **Struct**$*put* then invokes T$*put*. T$*put* does not find ***A*** in the map, so it outputs a header token at stream address 3, and invokes **array[int]**$*put*. **Array[int]**$*put* finds ***R*** in the map, and inserts a back reference to stream address 1 at stream address 4. At the receiving guardian, **array[int]**$*get* constructs an object ***R'*** from the encoded value of ***R***, and places ***R'*** in the decoding map. T$*get* extracts the next token from stream address 3, discovers it is a header token, and invokes **array[int]**$*get*. **Array[int]**$*get* discovers that the next token is a back reference, and returns ***R'*** from the map.

Fig. 27. The Get Operation Without Abstract Header Tokens

```
get = proc(cxt: dcontext, time: timeout)
        returns(T)
        signals(timeout)

    % Peek at first token.
    tok: token := dstream$peek(cxt.stream)
       resignal timeout
    if token$is_back_ref(tok) then
       addr: stream_addr := token$value_back_ref(tok)
       if  dmap$seen[T](cxt.dmap, addr) then
          % Object is old, remove token and look it up.
          dstream$extract(cxt.dstream)
          return(dmap$lookup[T](cxt.dmap, addr))
          end % if
       end % if

    % Object is new, create uninitialized version.
    u: ufo := ufo$make_empty(nil)
    ans: T := ufo_mask[T](u)

    % Enter the object in the initialization map.
    imap$enter[T](cxt.imap, ans)

    % Enter the object in the decoding map.
    addr: stream_addr := dstream$current(cxt.stream)
    dmap$enter[T](cxt.dmap, addr, ans)

    % Construct the external representation.
    y: xrep := xrep$get(cxt, time)
       resignal timeout
    ufo$change_uninitialized(u, y)
    return(ans)

    end get
```

The tokens produced by the optimized scheme appear in Figure 29. At the sending guardian, the only difference between the two schemes is that instead of placing a header token in the stream, T$*put* immediately invokes array[int]$*put*. At the receiving guardian, when T$*get* peeks at the token at stream address 3, it discovers the token is a back reference to steam address 1. When T$*get* looks up the stream address

Fig. 28. Tokens Produced With Abstract Headers

Stream Address	Token Type	Token Information
0	header	struct with two selectors
1	header	array with one element
2	data	int value
3	header	abstract T value
4	back reference	stream address 1

---

in the decoding map, it does not find an associated T object, so it invokes **array[int]$*get***. The latter proceeds as before.

4.2.4 Assumptions

To eliminate the need for careful prologues in the operations of an abstract type T, T objects must be decoded before they are referred to by other objects. This implies that values are decoded in an order such that all values used by T$*decode* are available when it is invoked. In the previous chapter, lazy decoding ensured this property by determining a legal order at run-time. For most types, a legal order can be determined statically, eliminating the need for lazy decoding and for careful prologues.

Fig. 29. Tokens Produced Without Abstract Headers

Stream Address	Token Type	Token Information
0	header	struct with two selectors
1	header	array with one element
2	data	int value
3	back reference	stream address 1

We divide implementations of data abstractions into two classes: *well-behaved* clusters are those whose objects may be completely decoded before being referred to; the *unpredictable* clusters are those for which run-time lazy decoding is required. The implementation described in the previous chapter protects objects of built-in type from premature access by decoding them before objects of abstract type. Similarly, the implementation developed in this chapter decodes values in two passes: values of built-in type and well-behaved abstract type are both decoded in the first pass, and values of unpredictable abstract type are decoded in the second pass.

4.2.5 Trivial Encode and Decode Operations

Clusters whose *encode* and *decode* operations perform type conversions on their arguments, but no other operations, form the simplest class of well-behaved clusters. We call such operations *trivial encode*'s and *decode*'s. When a type T has trivial *encode* and *decode* operations, the task of translating a T value into built-in values is simplified. If all the *encode* or *decode* operations invoked in the course of encoding or decoding a T value are trivial, then the translation task for T values is itself trivial, as it suffices to transmit the underlying representation of a T object as a built-in object.

If T is a type having trivial *encode* and *decode* operations, then there is no need to use lazy decoding for T values; furthermore, there is no need for the *put* and *get* operations to check for sharing. Normally, when *put* encounters a T object whose value has already been encoded in the current message context, it inserts a back reference token indicating the stream address where the encoded T value starts. Suppose T has a trivial *encode*, and that T$*put* invokes **xrep**$*put* without checking the encoding map. If **xrep**$*put* discovers that the **xrep** value has already been encoded, it inserts a back reference to the start of that encoded value. Since we have eliminated

abstract header tokens, the stream address of the encoded **xrep** value is the same as the stream address of the encoded T value, so it suffices to have the lower level *put* place the token in the stream.

An analogous argument suffices to show that T$*get* does not need to check for sharing, since any sharing that exists will be detected at a lower level. Furthermore, there is no need to create an uninitialized version of the T object, since the uninitialized version of its external representation will do as well. Finally, since no uninitialized versions of T objects are created, T$*initialize* does not need to iterate through the initialization map. In summary, the *put*, *get*, and *initialize* operations for T can be reduced to simple invocations of the *put*, *get* and *initialize* operations of T's external representation.

The compiler can easily detect trivial *encode* and *decode* operations. As a further optimization, the binder could replace the invocations of the trivial T$*put*, T$*get*, and T$*initialize* operations by direct invocations of the **xrep**$*put*, **xrep**$*get*, and **xrep**$*initialize* operations, eliminating levels of procedure linkage.

4.2.6 The External Type Closure

A second way to eliminate the need for lazy decoding is to recognize types that cannot have self-referential external representations. In this section we describe a fairly simple way to recognize statically that no objects of a type will require lazy decoding or uninitialized versions.

Let T and S be types. We define the *ET* (external type) relation among types in the following way:

If T is a primitive type, then there is no type S such that (T, S) ∈ *ET*.

If T is a composite type, then (T, S) ∈ *ET* if and only if S is a component type of T.

If T is an abstract type, then (T, S) ∈ *ET* if and only if S is T's external representation type

We use *ET*(T) to denote the set of types S such that (T, S) ∈ *ET*. For example,

ET(**string**) = ∅
ET(**oneof**[item: T, empty: **null**]) = {T, **null**}.

If set[T] is a parameterized, abstract type having as external representation type **sequence**[T], then:

ET(set[**int**]) = {**sequence**[**int**]}.

The *ETC* (*external type closure*) relation among types is defined to be the transitive closure of *ET*. Intuitively, *ETC*(T) is the set of types whose values will be included in a message containing a T value. The external type closure is similar to the concept of type closure found in [Atkinson 76].

ETC(**string**) = ∅
ETC(**oneof**[item: T, empty: **null**]) = {T, **null**} ∪ *ETC*(T)
ETC(set[**int**]) = {**sequence**[**int**], **int**}.

Before discussing the use of the external type closure, let us introduce some convenient terminology. For an abstract type T, we state that T is *recursively defined* if it belongs to its own external type closure. For example, we recall the int_list type introduced in Chapter Two, whose external representation is defined by:

```
xrep = record[car: int, cdr: link]
```

```
link = oneof [next: int_list, empty: null].
```

It is easy to verify that:

$$ETC(\text{int\_list}) = \{\textbf{int}, \text{int\_list}, \text{link}, \textbf{null}, \textbf{xrep}\},$$

where **xrep** and link are abbreviations for the **record** and **oneof** types. Since int_list is a member of its own external type closure, it is recursively defined.

We say that a procedure P *directly calls* procedure Q if the text of P contains an invocation of Q. We say that P *calls* Q if Q is in the transitive closure of P's "directly calls" relation.[1]

The basic claim we make in this section is that if a type is not recursively defined, then it does not require lazy decoding. It is possible to optimize the task of decoding values of such types in the following way. T$*get* may immediately decode a T object's external representation, as shown in Figure 30, rather than using a *ufo* to create an uninitialized object version. We will refer to this operation as the *simple get.*

Our argument that the simple *get* may be used for types that are not recursively defined takes the following form. To show that the simple *get* is safe for non-recursively defined types, we show that if the simple T$*get* attempts to use a value prematurely, then T must be recursively defined. This argument is presented in three steps:

---

1. The "calls" and "directly calls" relations are static: when we say that P calls Q, we do not mean that each *invocation* of P will cause an invocation of Q. For example, although the *get* operation for a **oneof** calls the *get* operations of all its component types, only one component *get* will actually be invoked by the **oneof**'s *get.*

Fig. 30. The Get Operation for a Non-Recursive Type

```
get = proc(cxt: dcontext)
        returns(T)
        signals(timeout)

    % Peek at first token.
    tok: token := dstream$peek(cxt.stream) resignal timeout
    if token$is_back_ref(tok) then
       addr: stream_addr := token$value_back_ref(tok)
       if  dmap$seen[T](cxt.dmap, addr) then
          % Object is old, remove token and look it up.
          dstream$extract(cxt.dstream)
          return(dmap$lookup[T](cxt.dmap, addr))
          end % if
       end % if

    % Object is new, remember steam address and decode xrep.
    addr: stream_addr := dstream$current(cxt.stream)

    % Construct and decode the external representation.
    y: xrep := xrep$get(cxt) resignal timeout
    x: T := T$decode(y)
    dmap$enter[T](cxt.dmap, addr, x)
    return(x)

    end get
```

Claim One: if T$*get* invokes T$*decode*, and the latter attempts to use the value of an S object, then S ∈ *ETC*(T).

Claim Two: if T$*get* invokes T$*decode*, and the latter fails when trying to use the value of an S object, then T ∈ *ETC*(S).

Claim Three: if S ∈ *ETC*(T), and T ∈ *ETC*(S), then T ∈ *ETC*(T).

To establish the first claim, we observe that for an S object to be accessible from T$*decode*, S$*get* must have been invoked by T$*get*, implying that T$*get* calls S$*get*. By inspecting the code for the *get* operations, one can see that T$*get* directly calls S$*get* if and only if S ∈ *ET*(T). It follows that T$*get* calls S$*get* if and only if S ∈ *ETC*(T).

To establish the second claim, we observe that an attempt to use the value of an uninitialized S object can fail only while the first S$*get* operation constructing it has been invoked but has not yet completed, for only then is the *ufo* representing the S object in the *empty* state. If T$*decode* can access an S object, then T$*get* must have been invoked by S$*get*, thus $T \in ETC(S)$.

We may illustrate this last point by recalling the cyclic engine and caboose types used as an example in Chapter Three. In that example, we traced in detail how an engine-caboose pair is decoded. Let us replace the usual caboose$*get* operation by a simple *get* operation, and briefly retrace the steps in the example. All goes well until the simple caboose$*get* invokes caboose$*decode*. The latter invokes engine$get_serial, which fails because the *ufo* representing the engine is in the *empty* state, since the engine$*get* operation constructing the engine object has been invoked, but has not yet terminated.

To establish the third claim, we make use of the fact that for all types T_1 and T_2:

$$T_1 \in ETC(T_2) \Rightarrow ETC(T_1) \subseteq ETC(T_2)$$

which follows directly from the definition of the *ETC* relation as a transitive closure. Therefore:

$$S \in ETC(T) \text{ and } T \in ETC(S) \Rightarrow T \in ETC(T).$$

Having established that $S \in ETC(T)$ (Claim 1), and $T \in ETC(S)$ (Claim 2), we therefore have $T \in ETC(T)$, demonstrating that T is recursively defined.

As a final remark on the simple T$*get* operation, we note that when decoding a T

object *A*, it is not necessary to enter an uninitialized version of *A* in the decoding map before *A*'s external representation is constructed. In the general case, an uninitialized version is placed in the map to catch cycles of reference. However no such cycles can exist when T is not recursively defined, for otherwise T$*get* calls T$*get*, and T $\in$ *ETC*(T).

The external type closure of a type T may be computed statically. By definition, external representations, unlike concrete representations, are the same at every guardian. Since the external type closure of a type T is defined entirely in terms of external representations, it is the same for all T implementations. Furthermore, we may assume that external representations are changed rarely, if at all, since changing a type's external representation requires modifying every implementation of that type in the system. This implies that once *ETC*(T) is computed, it is unlikely to change.

Since the external representation used by a type is known to the Library, it is a simple matter to compute the external type closure once the requisite specification information has been collected. The external type closure of an abstract type T should be part of the specification information about T maintained by the Library.

The cluster-dependent optimizations just described may interact with the distinction between careful and normal modules in the following way. If the compiler recognizes that a particular cluster is well-behaved, either because it has trivial *encode* and *decode* operations, or because it is not recursively defined, then it informs the Library of that fact. When the binder constructs a program, it extracts information about each module being bound from the Library. The binder does not need to insert careful prologues in the careful versions of operations of well-behaved clusters. Moreover, it is easy to detect the special case in which every module implementing a

type in *ETC*(T) is well-behaved, meaning that there is no need to use separate copies to distinguish between normal and careful versions of those modules.

4.2.7 The Function of the Binder

The *put* and *get* operations of an abstract type T can be constructed by the binder, since the only type-dependent aspect of *put* or *get* is the choice of external representation type.

Only the binder can determine whether an instantiation of an abstract type parameterized by type is recursively defined, since the parameterized type's external type closure cannot be determined without knowledge of the instantiated parameter type. For example, the set[T] abstraction described above has the following external type closure:

$$ETC(\text{set[T]}) = \{\text{sequence[T]}, T\} \cup ETC(T).$$

Thus, set[T] is recursively defined for all and only those types T such that set[T] is a member of *ETC*(T). For each instantiation, the binder can decide which *put* and *get* to use, and whether careful prologues are required. Like any other type, a parameterized type having trivial *encode* and *decode* operations does not require lazy decoding.

When binding the careful version of a T cluster, the binder decides whether to place careful prologues in the cluster operations, and which of the three kinds of *get* operations to use for T. The binder first checks whether T has trivial *encode* and *decode* operations. If so, invocations of T$*get* may be replaced by invocations of the *get* operation of T's external representation. If the *encode* and *decode* operations are non-trivial, the binder then checks whether T is recursively defined, using type

information in the Library, and information about instantiated type parameters. If T is not recursively defined, it can be given the simple *get* operation that directly invokes T$*decode* on the external representation. If either optimization applies, the careful versions of the T cluster operations are bound without careful prologues. If the T cluster has a non-trivial *decode*, and if T is recursively defined, then the general *get* operation must be used, and the binder must place careful prologues in the operations of the T cluster.

To make these decisions, the binder requires two kinds of information from the Library. To determine whether an abstract T is recursively defined, the library must maintain T's external representation type, and T's external type closure. The Library must also keep track of which T clusters have trivial *encode* and *decode* operations.

4.2.8 Optimizing The Initialization Stage

The initialization stage is another part of the translation task that can be optimized. One refinement suggests itself immediately: if the initialization map is empty at the end of the setup stage, there is no need to initialize object versions, or to remove *ufo*'s. It is only necessary to incur the expense of initialization and clean-up when uninitialized versions have actually been created.

We can also determine at binding-time that objects of a given type cannot contain *ufo*'s, requiring no initialization stage or cleanup traversal. If every type in a type T's external type closure is implemented by a well-behaved cluster, then there will be no *ufo*'s to initialize or remove. If *ETC*(T) contains no recursively defined types, the condition can be established statically from specification information in the Library. If *ETC*(T) does contain recursively defined types, then when particular implementations

of those types are chosen at binding-time, the binder can check whether those types have trivial *decode* operations. If we can determine, either statically or at binding-time, that objects of type T cannot contain *ufo*'s, then T$*initialize* can be replaced by a dummy procedure that simply returns.

4.3 Constructing and Transmitting Messages

In the previous section, we discussed ways to optimize the translation between abstract and built-in values that takes place both before and after the actual message transmission. In this section we discuss ways to optimize the construction and transmission of messages containing the built-in values. We are primarily interested in reducing the amount of storage required to send and receive messages.

When transmitting a very large message, we may reduce the amount of storage needed for buffering by transmitting information before the message is completely constructed. In the scheme described in the previous chapter, the tokens placed in an encoding stream comprise the transmitted message. Tokens are placed in the encoding stream as the object referred to by the send statement is traversed. The encoding stream abstraction has the property that a token can be transmitted any time after it has been inserted in the stream. The encoding stream cluster could be implemented to transmit the tokens as soon as a certain number have accumulated, perhaps asynchronously. Encoding streams allow storage use to be economized by interleaving value translation and message transmission. A disadvantage of this interleaving is that the receiver has no way to determine the size of a message before it is completely received.

In the special case where the communicating guardians reside on the same

machine, use the same language implementation, and where the implementation permits shared memory, message transmission can be accomplished quite easily. As we have stated before, the messages that are actually constructed and transmitted by the language implementation contain only values of built-in type. In the general case, a guardian wishing to transmit an integer value would encode that value into an integer data token. The receiver would then construct a new integer object from the received token. In the local case, the sender can just copy the integer directly into the receiver's address space, since both use the same representation for integers. Similarly, a guardian wishing to transmit the value of an **array[int]** could just copy the **array** into the receiver's address space. This scheme benefits both guardians: the sender may economize storage use, since it is not necessary to construct a message stream, and the receiver may economize processing, since it begins with a fully constructed representation object, instead of a stream of tokens that must be deciphered.

Now suppose the sender wishes to send a set[**int**], where set[T] is a parameterized abstract type having **sequence**[T] as its external representation. The sender can apply *encode* to the set[**int**], deriving a **sequence**[**int**]. The **sequence** can now be copied directly into the receiver's address space, where *decode* can be applied to construct a set[**int**] object.

Finally, suppose the sender wishes to send a set[set[**int**]]. The first application of *encode* returns a **sequence**[set[**int**]]. The next step is to create a new **sequence** by replacing each element with its external representation, deriving a **sequence**[**sequence**[**int**]]. Since this is an object of built-in type, it can be copied into the receiver's address space. By successive applications of *decode* operations, a copy of the original object is then reconstructed by the receiver.

These examples suggest how local message transmission can be optimized. The value of an object of built-in type is transmitted simply by copying that object into the receiver's address space. If the object is not of built-in type, it is reduced to built-in type by successively replacing abstract objects by their external representations, until no abstract objects remain. The resulting built-in object is then copied. The decoding process is the reverse of the encoding process; external representations are replaced by the abstract objects they represent. The remainder of this section describes the construction and interpretation of message objects in more detail.

We define the *message representation type* of a type T, denoted by MR(T), in the following way.[1]

> If T is primitive, MR(T) = T.
>
> If T is composite, then each component type is replaced by its message representation type, e.g. MR(**array**[S]) = **array**[MR(S)].
>
> If T is abstract, having external representation type XT, MR(T) = MR(XT).

We introduce *local_put* and *local_get* operations to construct message representations for objects. Since most of the structure of *local_put* and *local_get* operations is identical to the corresponding *put* and *get* operations, we will not describe them in great detail.

The *local_put* and *local_get* operations have the following interface specifications:

---

1. The message representation of a recursively defined type is a directly recursive type, which is not an expressible type in CLU.

```
local_put: proctype(T, map) returns(any)
local_get: proctype(any, map) returns(T)
```

For a type T, T$*local_put* accepts a T object as an argument, and returns an object of type MR(T), encoding the value of that argument. T$*local_get* accepts an object of type MR(T) as an argument, and returns a T object constructed from that argument.

All *local_put* and *local_get* operations check for sharing in the usual way. Map types similar to those used in the general scheme serve to detect sharing. Where the maps in the general scheme use stream addresses to refer to the encoded values of objects, the maps in the local scheme use standard object references.

The *local_put* and *local_get* operations for primitive types simply copy their arguments into the receiver's address space. The *local_put* operation for **array**[T] constructs an **array**[MR(T)] in the receiver's address space, where the latter object is constructed by replacing each **array**[T] element with the result of its *local_put* operation. The *local_get* operation constructs a new **array**[T] by replacing each element in the received **array**[MR(T)] with the results of its *local_get* operation. The *local_put* and *local_get* operations of the other composite types behave analogously.

The *local_put* for an abstract type returns the result of applying **xrep**$*local_put* to the argument's external representation. The *local_get* operation invokes **xrep**$*local_get* on its message argument.

If every type in the external type closure of a type T has a trivial *decode*, then the underlying representation of the T object is the MR(T) object, and there is no need to perform any translation.

Conclusions

In this chapter we evaluate our results, suggest some extensions, and list some areas for future research.

5.1 Summary and Evaluation

The scheme developed in this thesis is motivated by the claim that value transmission for programmer-defined types should be under the control of the programmer. As evidence for this claim, the introduction describes a number of situations in which the representations of values used within a guardian are inappropriate for communicating those values between guardians.

We propose the external representation scheme as a means for defining transmission. To evaluate the merits of this scheme, let us review the goals set forth in Chapter Two, and examine how we have met them.

Our first goal was to permit communicating guardians to use different implementations for a common data type, without causing a combinatorial growth in complexity as new implementations are developed. The external representation scheme accomplishes this goal by serving as an information-hiding mechanism. Since all guardians communicate by encoding information in a common external representation, no guardian depends on another's concrete representation, and the introduction of a new implementation is indistinguishable from duplication of an old implementation.

The ease of implementing and using a particular data type depends to a certain extent on the simplicity of its specification. We feel that the external representation

scheme provides a simple way to specify the meaning of transmission for a type. The specification for a programmer defined type T has two parts. The first step is to choose an external representation type XT, for which transmission is defined. The second step is to define abstract encoding and decoding operations, which translate between values of T and values of XT. Transmission is defined for T by the triple composition of the encoding operation, the previously defined transmission operation for XT, and the decoding operation.

Since the correctness of a type's implementation depends on correctly implementing the translation operations, the programs that perform the translation should be easy to locate and verify. The programmer implementing a transmissible type must provide *encode* and *decode* operations to translate between concrete and external representations. The input-output behavior of the *encode* and *decode* operations completely characterizes the translation process. To verify that transmission is implemented correctly, it suffices to verify the *encode* and *decode* operations.

The responsibility for message construction and interpretation is given to the language implementation, facilitating the task of the programmer.

Although the scheme can be used without mechanisms to preserve sharing structure and to transmit values of cyclic objects, we feel that the availability of such mechanisms is a major strength of our scheme. Later in this chapter we will compare our scheme to a simpler one that does not provide this kind of support.

Finally, we require that our scheme be acceptably efficient. Rather than attempt to define "acceptably efficient," let us examine the areas where efficiency may be an issue.

The first efficiency question we address concerns the expected complexity of the user-defined translation operations. We may assume that programmers will attempt to make the operations as efficient as possible. In particular, it seems reasonable to suppose that many transmissible types will be implemented having identical concrete and external representations, requiring trivial translation operations.

The sharing preservation mechanisms increase the amount of work to be done, since objects must be entered into and retrieved from maps. On the current CLU implementation, it is possible to compare object identity through a simple pointer comparison, meaning that standard hashing techniques can be used to make the map types quite efficient.

The mechanisms used to facilitate transmission of values of cyclic objects introduce a potential source of inefficiency in the form of an extra level of indirection in certain object references. This inefficiency can be reduced through a number of optimizations described in Chapter Four. A straightforward optimization permits us to restrict the run-time expense of using indirect references to certain procedure invocations, at a cost in storage. Slightly more complicated optimizations permit us to eliminate indirect references entirely for certain types, through the maintenance of relevant information in a library accessible both to the compiler and the binder.

Finally, there are several special cases that we expect to be common enough to optimize specially. By recognizing clusters using the same concrete and external representations, it is possible to make message construction and interpretation more efficient. When all the translation operations used to construct a message are trivial in this way, the expense of constructing or interpreting a message is comparable to copying the object whose value is being transmitted. When communicating guardians

reside on the same node, it is possible to reduce the work associated with message transmission to a significant degree by taking advantage of shared memory, as we discuss in Chapter Four.

5.2 Transmitting Untyped Objects

Our scheme may be extended to permit guardians to receive messages without decoding the contained values. For example, a *file server* guardian may provide reliable storage for information belonging to other guardians, without regard for the content of the information. In particular, it should be possible to store and retrieve the value of an abstract T object using such a server, even if the T type is not supported at the server's guardian. To provide this capability, we introduce an *image* type. An image object may be viewed as an undecoded message containing value of transmissible type. An image is constructed from a transmissible object using the same value encoding mechanism used to construct messages. The value decoding mechanism is used to reconstruct a copy of the object originally used to construct the image. Images are immutable and transmissible, and have the following operations.

```
encode_value: proctype[T: type](T) returns(image)
```

Encodes the value of *arg1* into the result.

```
decode_value: proctype[T: type](image) returns(T)
        signals(wrong_type)
```

Returns an object constructed from *arg1*.

Let ***A*** be an object of type T. The relation of images to transmission mechanisms can

be summarized as follows:

image$decode_value[T](image$encode_value[T](*A*)) ≡ T$*transmit*(*A*, message_context$create())

Images resemble CLU **any**'s, in that they are useful for managing objects independently of their types. However, there are several important differences between **any**'s and images. First of all, "**any**" describes the behavior of variables, not objects. Unlike image, **any** is not really an object type. Secondly, images are transmissible, while **any** does not have a *transmit* operation. Finally, there is no sharing between an image, and any other object. An object, an image created from it, and an object created from the image are all disjoint. By contrast, when an object is assigned to an **any**, and when that **any** is forced, the original object, the **any**, and the result of the **force** are identical.

Images can serve as a convenient way to store values on secondary storage. By making images storable, the same encoding and decoding operations can serve both for storage and transmission. Furthermore, the representation in storage of a value is independent of the concrete representation used by the creating guardian. A guardian may store an image constructed from a T object in secondary storage, change the concrete representation used by its T cluster, and still be able to retrieve the stored value (as long as T's external representation remains unchanged).

The most convenient way to encrypt values kept in secondary storage may be to provide the image type with encrypting operations, rather than providing each storable type with its own encrypting operation.

Images also provide a way to copy transmissible objects. An object may be

copied by encoding its value in an image, and then decoding the image. The result will be a completely disjoint object, having the same value as the original.

5.3 Implications of Own Data

The principal result of extending the communication primitives to a language including **own** data is to make the optimizations described in the previous chapter more difficult.

By distinguishing between modules that may encounter objects represented by *ufo*'s, and those that may not, we were able to restrict the execution of careful prologues. This optimization depends on our ability to guarantee that two conditions hold:

> No indirect references to objects exist while normal modules are executing (i.e. when a **receive** is not in progress).
>
> Only careful modules can execute while a **receive** is in progress.

Since normal and careful modules share **own** variables, unrestricted use of **own** variables may subvert the dichotomy between the two kinds of module versions. For example, the careful version of a module may store a reference to a *ufo* in an **own** variable, which may later be operated upon by the normal version, violating the first condition. Another kind of problem arises when a normal module stores a procedure in an **own** procedure variable. The careful version of the module may violate the second condition by invoking that procedure, supplying an indirect reference as an argument.

We can avoid these problems by brute-force methods, perhaps by traversing **own**

variables at the end of a **receive**, or by requiring that procedure variables always refer to careful versions of procedures. More refined methods undoubtedly exist, but their pursuit is best left to individual implementations.

5.4 Operation Extension by Overloading

Value transmission for an object is performed by the *transmit* operation of its type. The method used to provide an abstract type with a *transmit* operation differs significantly from the way abstract operations are usually provided in CLU. In this section we examine the reasons for this difference. In the following section, we suggest ways in which the method used to implement *transmit* may be generalized into a methodology for implementing other operations of abstract type.

Certain operations, such as *identical*, *copy*, and *transmit*, are useful to a wide variety of types. The language provides these operations for a collection of built-in types, and it is frequently useful to provide them for abstract types. We will identify three approaches to providing such operations. The first approach, which we call the *automatic* approach, is to have the language implementation provide the operation for the abstract type, usually in terms of the operations of the concrete representation type. The *identical* operation was defined in this way. In general, this approach is unsatisfactory, since the exact meaning of a type's operations (e.g., *copy*) depend on the abstraction, not on the type's implementation.

The second approach, which we shall call the *overloading* approach, is the one currently used in CLU. The language provides the built-in types with a collection of standard operations; the cluster implementing an abstract type may include procedures to implement the corresponding operations. The language requires that these

operations have standard interface specifications; for example, T$*copy* should have the form:

```
copy: proctype(T) returns(T)
```

CLU suggests guidelines for appropriately defining T$*copy*, although the language does not attempt to impose further restrictions either on the meaning or on the implementation of the operation.

In Chapter One, we observed that an abstract type's *transmit* operation cannot be provided automatically. One of the main conclusions of this thesis is that it is equally undesirable to provide abstract *transmit* operations by overloading. We claim that if users are given complete freedom to implement *transmit*, then the problems of sharing preservation and representation standardization remain unsolved, in any practical sense.

Let us briefly examine the problems that arise in an alternate scheme using overloading to provide abstract *transmit* operations. The *image stream* scheme used in the CLU reference manual to store values on secondary storage is used to construct messages. Image streams behave like the message streams used earlier in this thesis. All of the built-in types are given encoding operations to insert a value into an image stream, and decoding operations to extract a value from an image stream. Implementors of abstract types are expected to provide their types with encoding and decoding operations, constructed from the encoding and decoding operations of subsidiary types.

The first problem with the overloading scheme is that it is much more difficult to verify that information is being transmitted in the correct format. In any scheme,

communicating implementions of the same type must agree on an intermediate representation for values of the type. Using image streams, the compiler cannot check whether an encoding operation that may invoke a number of subsidiary encoding operations produces a correctly typed intermediate representation. On the other hand, the *transmit* operation permits static verification that the correct external representation type is used by a cluster, simply by type-checking the *encode* and *decode* operations. Of course, neither scheme can completely eliminate the possibility of error; however, the *transmit* scheme offers greater protection.

The second problem with the overloading scheme is the difficulty of preserving sharing. The encoding and decoding operations of the objects being sent must collect sharing information and encode it explicitly into the stream. One might think that the task could be facilitated by providing the programmer with access to encoding and decoding maps. In fact, we have considered many such schemes. Unfortunately, we have been unable to develop a scheme that did not seem excessively complicated and awkward.

Transmit is only one of a class of operations that are difficult to extend using overloading. We suggest *copy* as an example of another such operation. In CLU, the *copy* operation is intended to have the following effect:

> the *copy* operation should provide a "copy" of its input object, such that subsequent changes made to either the old or the new object do not affect the other. [Liskov 79, p.80]

Let us examine an abstract type whose *copy* operation does not readily lend itself to extension by operator overloading.

Consider a file system organized as a directed graph, where non-terminal nodes

are directories, and terminal nodes are files. A file is named by specifying a path from a distinguished *root* directory to the desired terminal node. Files and directories may be shared, since a given node may be accessible through one or more paths.

Consider the problem of defining and implementing a directory$*copy* operation that is to be used to create backup versions of directories. Given a directory, we wish to make a copy of the directed graph rooted at that directory. We use ***A'*** to denote the results of copying a graph node ***A***. We wish *copy* to preserve the sharing structure of this subgraph: i.e., if ***A***, ***B***, and ***C*** are nodes in the subgraph, and if ***B*** and ***C*** share a node ***A***, then ***B'*** and ***C'*** should share ***A'***.

These specifications cannot be implemented in a satisfactory manner using operator overloading. The problem is essentially that the user is given no way to detect non-local sharing structures. The directory$*copy* operation could conceivably be able to detect when a single directory has two links to the same file, but there is no straightforward way to detect that two distinct directories share a file. Furthermore, it is difficult to prevent the copy operation from recursing forever when it is applied to a subgraph containing cycles.

5.5 Operation Extension by Template

The third approach to operator extension, which we call the *template* approach, was used to provide abstract transmit operations. Using this approach, an operation provided for built-in types may be extended to abstract types, but the language imposes a rigid structure on the form of the operation's implementation.

For an abstract type T, we can informally describe the T$*transmit* operation in terms of the following five steps:

Step 1: Check for sharing.
Step 2: Encode the T value into its external representation.
Step 3: Transmit the external representation.
Step 4: Check for sharing.
Step 5: Decode the external representation into a T object.

Steps One and Two are performed at the sending guardian, while steps Four and Five are performed at the receiving guardian. The language controls the *form* of *transmit*, while the user controls its *meaning* through the provision of the *encode* and *decode* operations used in steps Two and Five.

In the remainder of this section, we will examine how this approach can be generalized to extend an arbitrary operation, and we will review a number of operations whose implementations are better effected by using templates than by using overloading.

We assume that some collection of built-in types and type constructors is provided with an *op* operation. For each such type S, S$*op* has the following interface specification:

op: **proctype**(AT_1,...,AT_n) **returns**(RT_1,...,RT_m) **signals**(...)

where each argument type and each result type (both normal and exceptional) is either a built-in type, or S. We use I to denote the set of indices i such that AT_i = S, and J to denote the set of indices j such that RT_j = S.

To extend the *op* operation to an abstract type T, the T cluster must provide *translation* operations, denoted here by T$*op_encode* and T$*op_decode*. The *op_encode* operation encodes the value of an argument of type T into a value of a *special representation* type ST, where ST has an *op* operation. The *op_decode* operation

accepts an argument of type ST, and returns a result of type T.

```
op_encode: proctype(T) returns(ST)
        signals(encode_error(string))
op_decode: proctype(ST) returns(T)
        signals(decode_error(string))
```

T$*op* is defined in terms of ST$*op* in the following way. An invocation such as

$$y_1,\ldots,y_m := T\$op(x_1,\ldots,x_n)$$

causes the invocation of:

$$y_1',\ldots,y_m' := ST\$op(x_1',\ldots,x_n')$$

where the values of the arguments to ST$*op* are defined by:

$x_i' = T\$op\_encode(x_i)$ for $i \in I$.
$x_i' = x_i$ otherwise.

The translation between the arguments to T$*op* and the arguments to ST$*op* is also sensitive to sharing, in the following way. All invocations of *op* take place with respect to a given *context*, where a context is analogous to the message context defined in Chapter Two. The scope of a context is defined as follows. When T$*op* is invoked directly from a user program, a new context is created. When an invocation of T$*op* causes the invocation of ST$*op*, the latter occurs with respect to the same context as the former. For all invocations of T$*op* occurring with respect to the same context, the following condition holds: if two arguments to T$*op* share a T object ***A***, then the corresponding arguments to ST$*op* will share a ST object ***A'***, where ***A'*** is constructed from ***A*** by a single application of T$*op_encode*.

If ST$*op* returns normally, then T$*op* returns normally, and the values of its results are defined by:

y_j = T$*op_decode*($y_j$') for j ∈ J.
y_j = y_j' otherwise.

Sharing among the results is preserved in the same way as sharing among the arguments: for all invocations of ST$*op* occurring with respect to the same context, if two results of ST$*op* share a ST object ***B'***, then the corresponding results of T$*op* will share a T object ***B***, where ***B*** is constructed from ***B'*** by a single application of T$*op_decode*.

If ST$*op* raises an exception, then T$*op* raises the same exception, and any objects returned by the exceptions are treated as results; i.e., if ST$*op*'s exception returns a ST object, then T$*op*'s exception returns a T object constructed from the corresponding ST object by an application of T$*op_decode*. Finally, if *op_encode* or *op_decode* signal an exception, then T$*op* signals that same exception.

Templates are useful for defining operations that are sensitive to sharing structure. Since the *op_encode* and *op_decode* operations associated with such an operation are applied by the language implementation, not by user programs, the language implementation can do the bookkeeping required to recognize and keep track of sharing. As we have repeatedly argued in the case of the *transmit* operation, this kind of bookkeeping is tedious and error-prone if performed by the user.

Template definition may be viewed as a control abstraction; the cluster writer who defines an operation using a template definition need not be concerned with the mechanical details of sharing preservation, but the fact that sharing is preserved may be quite important. The programmer is free to concentrate on the individual translation operations, while the language implementation ensures that they are applied correctly.

5.5.1 Revising Standard CLU Operations

The first examples we will examine are standard CLU operations. As illustrated in a previous section, the problem of sharing preservation makes the *copy* operation difficult to extend satisfactorily using overloading. By using a template structured *copy* operation, the language implementation can detect sharing, while the meaning of the operation can be controlled by user-defined *copy_encode* and *copy_decode* operations.

For some types, *copy* will just copy the underlying concrete representation object. In that case, *copy_encode* and *copy_decode* may just perform **up** and **down** conversions. As an example of a type requiring more sophisticated translation operations, consider a PT (protected T) object consisting of a T object protected by an associated semaphore. When the PT object is copied, it would make no sense to copy the state of the semaphore, which may contain a collection of waiting processes. The PT$*copy_encode* operation returns the T component without the associated semaphore, while the *copy_decode* operation accepts a T object, creates a new semaphore, and then combines them to construct a PT object.

CLU's *similar* operation is used to determine when two objects of the same type have the same information content. Precisely what constitutes the interesting "information content" of an object is quite type-dependent. For instance, **array**[T]$*similar* is defined to check whether the two arrays being compared have the same bounds. If so, then T$*similar* is used to test pairs of corresponding elements for similarity. If all of these tests succeed, then the two arrays are deemed to be similar.

The definition of **array**[T]$similar could be altered to encompass the sharing structures of the arrays being compared. Two objects may be compared as directed

graphs of objects, where nodes represent component objects, and edges represent logical containment. Let us define a *globally_similar* operation for the built-in types to test for *similar* objects having the same structure as directed graphs. Individual node similarity is tested in the usual manner.

```
globally_similar: proctype(T, T) returns(bool)
```

Global sharing structure is recognized by accumulating a table of corresponding components of the objects being compared. If at any time, a component of one object corresponds to more than one component of the other, then the objects are not *globally_similar.*

We observe that since *globally_similar* returns no objects of T type, there is no need for a decoding translation operation.

When comparing the values of objects of the protected T type introduced above, let us assume we only wish to compare the values of the T components; we do not wish to compare the states of the associated semaphores. Under this assumption, the encoding translation operation only needs to extract and return the T component of its PT argument.

5.5.2 I/O Operations

We have observed that template definition imposes a rigid structure on the form of an operation's implementation. A benefit of this rigidity is that it becomes possible to use template structured operations to define interfaces between autonomous domains such as guardians. We have already seen how the structure of the *transmit* operation permits a division of labor between the communicating guardians, and

between the language implementation and the cluster writer. A large class of operations that not only involve sharing detection, but that require a degree of standardization among autonomous guardians, are operations to perform input or output activities using the values of abstract objects.

The first I/O operations we will examine are used to store and retrieve the values of objects on secondary storage. Let us define *store* and *retrieve* operations for the built-in types, having the following interface specifications:

```
store: proc(T) returns(file_name)
retrieve: proc(file_name) returns(T)
```

Mechanically copying objects' concrete representations to secondary storage is not a satisfactory way to implement *store* and *retrieve*. To illustrate this point, we recall the protected T type. When storing the value of a protected T object, it makes little sense to store the state of the associated semaphore. Similarly, overloading is not a satisfactory way to implement *store* and *retrieve*, for two reasons. First, we would like to control how sharing structure is preserved. Second, we would like to use static type-checking to ensure that values of a type are stored in a standard format, since we would like to share stored values with other guardians that might use different concrete representations for the type.

We may extend *store* and *retrieve* to abstract types by selecting for each abstract type T, a *stable representation* type ST, with appropriate translation operations. We recall that by using a standard external representation, T values could be communicated between different implementations of T. Similarly, the use of a standard stable representation permits different implementations of T to store and retrieve one another's values. This may be particularly useful when replacing one

version of the T cluster by another; by leaving the stable representation unchanged, the new version can read values previously stored by old versions.

Another operation that should be sensitive to sharing structure is the *display* operation to display values of objects to humans. *Display* requires an encoding translation operation, but no decoding translation operation. The *display* operation is particularly useful for debugging. When debugging a program that uses a data abstraction T, the best way to display a T object's value is not necessarily to display the value of its representation. For instance, when debugging a program that uses a symbol table, a simple display of associated key-item pairs will be more useful than a more complicated display of hash tables and list structures. This kind of display is particularly appropriate for remote debugging, where an object of interest resides on a foreign guardian using a concrete representation unknown to the debugger. On the other hand, when debugging the symbol table *cluster*, the value of the representation is of interest.

We do not intend to explore the difficult question of how values are to be represented to users; however, one could imagine displaying an object's value as a directed graph on a high resolution cathode-ray screen. The built-in types and type constructors may be given a standard display representation, which may be extended to abstract types by selecting for each abstract type T, a *display representation* type DT, with a translation operation from T to DT. The inverse translation from DT to T might be used to define T literals.

5.5.3 Conclusions

Operation extension by template definition appears to have two advantages. It serves to implement sharing-sensitive operations for abstract types in a way that is not currently possible in CLU. Furthermore, template definition eases the standardization problems that arise in a distributed system; although we cannot guarantee that the information being released by *transmit*, *store*, or *display* is correct, we can guarantee that it is in the correct format.

When defining template operations that operate on cyclic objects, one encounters the same problems we encountered earlier with self-referential external representations. If we make the same choice we made for *transmit*, we may operate on arbitrary cyclic objects by imposing restrictions on *op_decode* operations. The language implementation must then introduce uninitialized object versions in the manner described above.

On the negative side, there may be an efficiency penalty to having the language implementation apply translation operations and check for sharing. A programmer having semantic information about an abstraction can detect optimizations that the language implementation cannot. By expending more human effort, in is undoubtedly possible to improve individual implementations. There is a characteristic trade-off between the increased convenience and reliability provided by template-structured operations, and the ability to construct optimizations on an individual basis provided by overloaded operations.

5.6 Applicability to Other Languages

Since we have presented our communication primitives as an extension to CLU, it is natural to ask how readily our primitives can be adapted to other languages.

One aspect of CLU that is essential to our scheme is the notion of data abstraction. One of the principal motivations is the belief that different representations of information are appropriate for different purposes. The representation used to transmit a value between guardians may be different from the representation used within a particular guardian, and different representations for objects of a type may used at different guardians. If the language contains no facilities for encapsulating representation information, then communication among differing implementations must be based on voluntary conventions, not on language features.

The fact that CLU is an object-oriented language, as opposed to a variable-oriented language, is not crucial to our scheme. Although we have spent much of our effort defining the effects of transmission on sharing structure, the same problems arise in languages having explicit reference types, and the same solutions are applicable.

5.7 Directions for Further Research

Defining value transmission is only the first of many difficult problems in the development of communication primitives for a distributed application language. A comprehensive survey of the outstanding research areas in this field could easily fill another chapter; accordingly, we mention only those questions that arise directly from this research.

Rather than limit messages to the value of a single object, it may be convenient to introduce explicit message types. One possibility is to define a message type as consisting of a tag followed by objects whose values are transmitted together. Port types would consist of a list of message types. Examples of message types are:

```
employee(name: string, salary: int)
error(message: string)
```

If two objects whose values are sent in a message share a component, it must be decided whether the objects constructed by the receiver should also share. If that effect is desired, all the objects in a message should be encoded and decoded in the same message context. Alternatively, if the opposite effect is desired, a distinct message context should be used for each object.

An alternative to explicit message passing is to support inter-guardian communication by remote procedure invocation. The value transmission mechanisms developed here can be used to pass arguments from the invoking guardian to the guardian where the requested action is carried out, and to return any results. This kind of remote invocation differs from usual procedure invocation in CLU, where procedures pass arguments by sharing objects between the caller and the called procedures. Remote argument passing resembles traditional call-by-value schemes. We feel that value transmission is better suited to remote invocation, as node failures and inherent unreliability in the communication medium can cause remote invocations to fail in ways that are not possible for local invocations.

In summary, the value transmission scheme developed here can be adapted to a number of different communication primitives. Determining the best scheme (or schemes) to incorporate into a language is an area that would benefit from further

research.

The **send** and **receive** statements used in this thesis were defined as simply as possible. Such simple **send** and **receive** statements are probably not the best choice of primitives. Actual language primitives would probably have to be more sophisticated, and would certainly have to address issues that we have avoided. For example, it may be useful to provide primitives to support patterns of communication, such as remote procedure invocation, paired requests and responses, or forwarding of requests to other guardians. More research is needed to determine which of these patterns, if any, should be supported in a higher-level language.

We have made no mention of the degree of reliability provided by the **send** and **receive** primitives. The **send** primitive may or may not attempt to retransmit messages that appear to have been lost, and it may or may not cause the same message to be received more than once. The degree of reliability built into a primitive undoubtedly depends on its form; a remote invocation primitive would have to be fairly reliable, while a simple **send** need not be. The inherent unreliability of a distributed system may complicate the programmer's task; the degree to which the proper choice of communication primitives may ease such problems is an important area for future research.

We have used ports to indicate the destination of messages, and to insure type correctness. We have not addressed how ports are acquired, or whether ports are really the best way for guardians to name one another. The question of inter-guardian naming depends on assumptions about the organizations of programs, and the organizations of guardians.

We have not given a formal semantics for value transmission. A number of approaches to formal description of object-oriented languages exist [Berzins 79, Schaffert 78, Scheifler 78]; it would be interesting to extend these descriptions to value transmission.

The scheme developed in this thesis permits guardians to change the concrete representation used for a type without that change being visible outside the guardian. We have not provided any easy way to change the external representation used by an abstraction, as such a change requires changing implementations at all guardians supporting the type. Changing a type's external representation is a special case of the general problem of replacing programs in a distributed system.

Finally, we have noted that the template scheme used to implement and define *transmit* can be extended in a very straightforward manner to implement and define such operations as *copy*, *similar*, *store* and *retrieve*, and *display*. It is natural to enquire whether other operations may be defined in this way, and whether other kinds of templates may be useful for defining other operations.

This empty page was substituted for a blank page in the original document.

References

[Atkinson 76] R. Atkinson, "Optimization Techniques for a Structured Programming Language," S.M thesis, Massachusetts Institute of Technology, May 1976.

[Berzins 79] V. Berzins, "Abstract Model Specifications for Data Abstractions," M.I.T. Laboratory for Computer Science TR 221, July 1979.

[Crocker 75] S. D. Crocker, "The National Software Works: A New Method for Providing Software Development Tools Using the ARPANET," Proc. Meeting on 20 Years of Computer Science, Pisa, Italy, July 1975.

[Fabry 76] R. S. Fabry, "How to Design a System in Which Modules can be Changed on the Fly," Proceeding of the Second International Conference on Software Engineering, San Francisco CA, October 1976, pp. 470-477.

[Feldman 79] J. Feldman, "High Level Programming for Distributed Computing," CACM 22, 6, June 1979, pp. 353-367.

[Friedman 76] D. P. Friedman and D. S. Wise, "CONS Should Not Evaluate its Arguments," In S. Michaelson and R. Milnor (eds), *Automata, Languages and Programming*, Edinburgh University Press, Edinburgh 1976, pp. 257-284.

[Gligor 79] V. D. Gligor and B. G. Lindsay, "Object Migration and Authentication," IEEE Transactions on Software Engineering, Volume SE-5, 6, pp. 607-611.

[Haber 78] N. Habermann, "Dynamically Modifiable Distributed Systems," Proceedings of the Distributed Sensor Net Workshop, Carnegie-Mellon University, Pittsburgh PA, December, 1978, pp. 111-114.

[Hender 76] P. Henderson and J. H. Morris, "A Lazy Evaluator," Proceedings of the Third ACM Symposium on Principals of Programming Languages, 1976, pp. 95-103.

[Levine 78] P. Levine, "Facilitating Interprocess Communication in a Heterogeneous Network Environment," M.I.T. Laboratory for Computer Science TR 184, July 1977.

[Liskov 79] B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, B. Scheifler, A. Snyder, CLU Reference Manual, M.I.T. Laboratory for Computer Science TR 225, October 1979.

[Liskov 79a] B. Liskov, "Primitives for Distributed Computing," Proceedings of the Seventh Symposium on Operating Systems Principals, Pacific Grove CA, December 1979, pp. 33-43.

[Neigus 73] N. J. Neigus, File Transfer Protocol, NIC \17759, August 1973.

[Postel 74] J. Postel, "NSW Protocols Version 2," Stanford Research Institute, 1974.

[Reed 78] D. Reed, "Naming and Synchronization in a Decentralized Computer System," M.I.T. Laboratory for Computer Science TR 205, September 1978.

[Schaffert 78] J. C. Schaffert, "A Formal Definition of CLU," M.I.T. Laboratory for Computer Science TR 193, January 1978.

[Scheifler 78] R. W. Scheifler, "A Denotational Semantics of CLU," M.I.T. Laboratory for Computer Science TR 201, May 1978.

[Snyder 79] A. Snyder "A Machine Architecture to Support an Object-Oriented Language," M.I.T. Laboratory for Computer Science TR 209, March 1979.

[Sollins 79] K. Sollins, "Copying Complex Structures In a Distributed System," M.I.T. Laboratory for Computer Science TR 219, May 1979.

[Svobod 79] L. Svobodova, B. Liskov, D. Clark, "Distributed Computer Systems: Structure and Semantics" M.I.T. Laboratory for Computer Science TR 215, March 1979.

[Telnet 73] Telnet Protocol Specification, NIC \18639, August 1973.

[White 74] J. White, "The Procedure Call Protocol Version 2," Stanford Research Institute, 1974.

[Wulf 76] W. A. Wulf, R. L. London, M. Shaw, "Abstraction and Verification in Alphard: Introduction to Language and Methodology," Carnegie-Mellon University and USC Information Sciences Institute Tech. Reports, 1976.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: MIT/LCS/TR-234
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): Transmitting Abstract Values in Messages

5. TYPE OF REPORT & PERIOD COVERED: M.S.Thesis-April 25, 1980

6. PERFORMING ORG. REPORT NUMBER: MIT/LCS/TR-234

7. AUTHOR(s): Maurice P. Herlihy

8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0661 MCS74-21892 A01

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
MIT/Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS:
ARPA/Dept. of Defense
1400 Wilson Boulevard
Arlington, VA 22209

/NSF/Associate Prog. Director/Office of Computing Activites
Washington, DC 20550

12. REPORT DATE: May 1980

13. NUMBER OF PAGES: 123

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):
ONR/Department of the Navy
Information Systems Program
Arlingon, VA 22217

15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report):
This document has been approved for public release and sale; its distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report):

18. SUPPLEMENTARY NOTES:

19. KEY WORDS (Continue on reverse side if necessary and identify by block number):
Abstract types
Distributed Systems
Message Passing
Modularity
Object-Oriented Programming
Programming Languages
Programming Methodology

20. ABSTRACT (Continue on reverse side if necessary and identify by block number):
This thesis develops primitives for a programming language intended for use in a distributed computer system where individual nodes may have different hardware or software configurations. Our primitives are presented as extensions to the CLU language. We assume that differences in hardware and in administrative policy require that individual nodes be free to choose their own local representations for common types, including user-defined types. Our main objective is to provide primitives to communicate values of user-defined type. Our primitives support a large degree of node autonomy, without requiring

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. that communicating nodes have prior knowledge of one another's special characteristics. We argue that the precise meaning of value transmission is type-dependent; thus the user, not the language, must control the meaning of transmission for values of a type.