T-SE/13/1/11363

Distribution and Abstract Types in Emerald

A. Black
N. Hutchinson
E. Jul
H. Levy
L. Carter

Reprinted from: IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. SE-13, No. 1, JAN. 1987.

Distribution and Abstract Types in Emerald

ANDREW BLACK, NORMAN HUTCHINSON, ERIC JUL, HENRY LEVY, AND LARRY CARTER

Abstract—**Emerald is an object-based language for programming distributed subsystems and applications. Its novel features include 1) a single object model that is used both for programming in the small and in the large, 2) support for abstract types, and 3) an explicit notion of object location and mobility. This paper outlines the goals of Emerald, relates Emerald to previous work, and describes its type system and distribution support. We are currently constructing a prototype implementation of Emerald.**

Index Terms—**Abstract data types, distributed operating system, distributed programming, object-oriented programming, process migration, type checking.**

I. Introduction

WHILE distributed systems are now commonplace, the programming of distributed applications is still somewhat of a black art. We believe that the complexity of distributed applications is heightened by the lack of programming language support for distribution. For example, most distributed applications are implemented by calling operating system communications primitives, such as send and receive. The programmer is responsible for locating the communications target, explicitly packaging parameters, and so on. Before the introduction of concurrent programming languages, concurrent programs were constructed in a similar fashion. Language support for concurrency greatly simplified concurrent programming; we believe that language support for distribution can have a similar effect on distributed programming. Experience with the remote procedure call facilities of Cedar/Mesa [2] and with the Eden Programming Language [1] has justified this belief. With Emerald, we intend to go beyond simple syntactic support for message send and receive, and address some of the fundamental semantic problems of distribution.

Although distribution has many benefits [22], it also introduces challenges for the designer of a distributed language. First, the language must present a model of distributed computation; it must provide the conceptual framework that allows the programmer to define the objects that he manipulates in both the local and distributed environment. Second, it must provide for both intra- and internode communication in an efficient manner. The semantics of communication and computation should be consistent in the local and remote cases. Third, it must allow the programmer to exploit the inherent parallelism and availability of a distributed system. Fourth, since shutting down and recompiling an entire distributed system in order to modify some component is unacceptable, the language must permit system extensibility without recompilation; existing programs must continue to work in collaboration with new programs.

Our research focuses on simplifying the programming of distributed subsystems and applications by providing language support for distribution. We have designed an object-based language, called *Emerald*, and a distributed run-time system for Emerald that facilitate the construction of distributed programs for a local area network of independent nodes (workstations). The novel features of Emerald include: 1) a single object model that is used for both programming in the small and in the large, 2) support for abstract types, and 3) an explicit notion of object location and mobility. The goal of our research is to demonstrate the feasibility of using one simple semantic model for programming both sequential, single-node applications, and concurrent, potentially distributed applications. We currently have a prototype Emerald compiler and run-time system running on a local area network of VAX® workstations.

The next sections present a discussion of previous work in distributed programming languages and an overview of Emerald. Following sections describe the type system and the support for distribution.

II. Review of Previous Systems

To date, languages have supported distribution in several different ways. In the Xerox Cedar System [33], a remote procedure call facility allows programs to access remote servers through standard Cedar/Mesa procedure calls [2]. The advantage of this approach is that it requires no change to the semantics of the language. Automatically generated stub routines on the client and server machines are responsible for packing and unpacking parameters and transmitting and receiving messages. Programmers access a remote service in the same way that they would access a local service, except that they must explicitly locate and connect to the service before it can be used.

Manuscript received January 31, 1986; revised June 16, 1986. This work was supported in part by the National Science Foundation under Grants MCS-8004111 and DCR-8420945, by the University of Copenhagen, Denmark, under Grant J.nr. 574-2,2, and by a Digital Equipment Corporation External Research Grant.

A. Black, N. Hutchinson, E. Jul, and H. Levy are with the Department of Computer Science, University of Washington, Seattle, WA 98195.

L. Carter is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598. This work was performed while he was a visitor at the University of Washington.

IEEE Log Number 8611363.

®VAX is a registered trademark of Digital Equipment Corporation.

TABLE I
EMERALD LANGUAGE FEATURES

	Emerald	Argus		Xerox RPC	Eden		Smalltalk
		CLU	Guardians		EPL	Objects	
Uniform Object Model	√	√					√
Distribution	√		√	√		√	
Mobile Objects	√					√	
Direct Objects	√	√		√	√		√
Concurrency Control	√		√	√	√	√	
Type Checking	√	√	√	√	√		
Abstract Types	√			√			

At the University of Washington, the Eden Programming Language (EPL) [3] has been developed for writing distributed applications on the Eden system [1]. EPL is an extension of Concurrent Euclid [19] that provides location-independent invocation of Eden's objects in an integrated distributed system. Because the location of an Eden object is (conceptually) evaluated at each invocation, objects are free to move at any time; one need not locate or connect to an object before invoking it.

The M.I.T. Argus system [24], [25] is an ambitious distributed language project that extends the CLU language [23] to support atomic transactions in a distributed environment. An Argus *Guardian* encapsulates the notion of a physical machine. Inside a guardian are data objects and processes. One Guardian communicates with another by calling a handler in the target Guardian, to which data are passed by value.

In general, object-based systems and languages have viewed their objects in two ways: as large, long-lived resources (e.g., files) as in operating systems such as Hydra [35] and StarOS [20], or as small resources (e.g., records and integers), as in languages such as CLU [23] and Smalltalk [15]. In a distributed environment, both views seem to have their place; the Argus and EPL languages each support two kinds of objects. Argus has Guardians, which are network-wide objects, and CLU objects, which are local to a Guardian; EPL has network-wide Eden objects that contain local EPL variables, monitors, and modules. The reason for this dichotomy is one of locality and performance; local objects communicate through shared store, while network objects communicate through message passing, which requires more communications overhead. Unfortunately, this requires the programmer to use two different object abstraction mechanisms, to code in two different styles, and to foresee all possible uses to which an object will be put. For example, while programming a Collaborative Editing System in Argus, Greif *et al.* [16] have observed that a designer can be forced to use a Guardian where a cluster might be more appropriate.

Emerald has drawn on the experience of all of these systems. The most important difference between Emerald and these systems is Emerald's uniform model of computation. Like Smalltalk, all entities in Emerald are objects, and a single semantic model suffices to define them. Unlike Smalltalk, however, Emerald is a distributed programming language; its object model is sufficient to describe both local data objects and potentially remote objects containing independent processes. Table I enumerates the principal features of Emerald and compares them to those of Argus, Xerox RPC, EPL, and Smalltalk. The following section provides a brief overview of the Emerald programming language, focusing on its uniform object model, and following sections deal with two important aspects of Emerald: its type system which is based on the concept of abstract types, and its support for distribution.

III. INTRODUCTION TO EMERALD

Emerald is object-based and all information is encapsulated in objects. An object model is appropriate for a distributed system because it implicitly defines 1) the units of distribution and movement, and 2) the entities that communicate. All objects in Emerald are coded using the same object definition mechanism, regardless of the way in which they will be used. The Emerald object model is appropriate for defining *small* objects such as integers, characters, and Booleans as well as *large* objects such as directories and compilers. While different objects may be represented by the system in different ways, all objects exhibit the same semantics. Each Emerald object exports a set of operations; an object can be manipulated only by invocation of one of those operations. Furthermore, Emerald objects are *mobile*. Objects can move at any time, and can be invoked without knowledge of their location.

Each Emerald object has four components:

1) An *identity*, which distinguishes the object from all others within the network.

2) A *representation*, which consists of the data stored in the object. The representation of a programmer-defined object is composed of a collection of references to other objects.

3) A set of *operations*, which defines the functions and procedures that the object can execute. Some operations are exported and may be invoked by other objects, while others may be private to the object.

4) An optional *process*, which operates in parallel with

```
const inCoreFile == object inCoreFile
  export Read, Seek, Write
  monitor
    const maximumSize == 200
    const CharacterVector == Vector.of[Character]
    var contents : CharacterVector
    var position : Integer

    operation Read → [c : Character]
      if position <= contents.upperbound then
        c ← contents.getElement[position]
        position ← position + 1
      else
        c ← nil
      end if
    end Read

    operation Seek[p : Integer]
      position ← p
    end Seek

    operation Write[c : Character]
      assert position <= contents.upperbound
      contents.setElement[position, c]
      position ← position + 1
    end Write

    initially
      contents ← CharacterVector.create[maximumSize]
      position ← 0
    end initially
  end monitor
end inCoreFile
```

Fig. 1. An inCoreFile object.

invocations of the object's operations. An object with a process has an active existence and executes independently of other objects. An object that has no process is a passive data object and executes only as a result of invocations.

An Emerald object also has several attributes. An object has a *location* that specifies the node on which that object is currently resident. Emerald objects can be defined to be *immutable*. Immutability is an assertion on the part of the programmer that the abstract state of an object does not change; it is not a concrete property and the system does not attempt to check it. On the other hand, the system takes advantage of immutability by copying such objects on remote reference.

Fig. 1 shows an Emerald definition of a simplified file object. This object supports the usual read, seek, and write operations expected from files, but of only a single character at a time. Its representation consists of a vector of characters and a current position indicator. Its three operations, *Read*, *Seek*, and *Write*, are exported and therefore available to users of the object. Since inCoreFile has no process, it is a passive data object and executes only as a result of invocations.

To exploit the inherent parallelism of distributed systems, Emerald supports concurrency both between objects and within an object. Separate threads of control are provided in the form of processes. Each object may have a *process section* specifying a parameterless, anonymous operation to be invoked asynchronously when the object has been initialized. Processes on the same processor execute in quasi-parallel with respect to each other and concurrently with respect to processes located on other processors.

While an object has a single independent process, at any point in time multiple processes can be executing within a single object. This results from multiple invocations of an object's operations by other processes. Operations and variables may optionally be specified in a *monitor section* of the object. Processes executing monitored operations have exclusive access to the monitored variables and may synchronize using conventional *condition* variables [18]. An object's process normally executes outside the monitored section, but, like any other process, it can invoke the monitored operations should it need access to shared variables.

Each object has an optional *initially section*—a parameterless operation that executes exactly once when the object is created and is used to initialize the object's state. When the initially operation is complete, the object's process is started and invocations can be accepted.

Although Emerald has a single, uniform model of objects, objects are implemented in several different ways. An important goal of Emerald is to provide a very efficient implementation for objects. In order to accomplish this goal, the Emerald compiler chooses for each object an implementation style appropriate for its use. Some support of this type is available in CLU [31]. The Smalltalk language also has two implementation styles: one for primitive objects such as integers and arrays, and one for user-defined objects. While primitive object operations are relatively efficient in Smalltalk, there is no mechanism whereby users can define equally efficient types.

There are three styles of implementation available for Emerald objects. Standard types such as integer are usually implemented by a single word of storage and compiler-generated in-line operations. For example, the integer *add* operation is reduced to a single machine instruction. Objects that are local to another object can often be implemented by a compiler-allocated data area and their operations implemented as normal (Pascal-like) procedure calls, thus avoiding the more general invocation mechanism provided by the run-time kernel. Finally, objects that can move or be remotely referenced are implemented by a kernel-allocated data area and are indirectly referenced through an object table. Invocations of such objects are performed by a combination of compiled code and the run-time kernel. For example, if inCoreFile is used as a temporary file within another object it may be represented in storage with local pointers and manipulated by in-line code. If it is used to convey information between passes of a distributed compiler it will require a remote procedure call interface.

IV. Types in Emerald

Emerald is a statically typed language that supports abstract types. We first motivate our decision to make Emerald a typed language (Section IV-A) and our decision to support abstract types (Section IV-B). The *conformity* relation which forms the basis of the type system of Emer-

ald is then introduced, first informally in Section IV-C, and then rigorously in Section IV-D. Section IV-E compares our type system to that of other languages and provides reasons for the major differences, and Section IV-F discusses type system implementation considerations.

A. Why Types?

In standard *value-oriented* programming languages, the type system protects the user from misinterpretation of values, e.g., it prevents a character operation from being applied to an integer. In an object-oriented language the programmer is not allowed to apply operations to values at all; he may request only that an object perform an operation on *itself*. The point here is that the representation of the object is never handled directly by external operations; only the internal operations of the object are allowed access to its representation. As a consequence, neither Smalltalk objects, nor Eden objects, nor Emerald objects need be typed in order to guarantee their integrity. In contrast to Smalltalk, where identifiers do not have declared types, we have chosen to make Emerald a typed language, i.e., the programmer must associate a type with each identifier that she declares. We expect to see several advantages from this approach.

First is better detection and notification of programming errors. Many errors found in Smalltalk programming are of the "message not understood" variety. In Smalltalk this message can be generated only at run time, when an invocation is attempted on an object that does not implement it; it generally indicates that a type error has occurred. In other words, the object assigned to a variable is not of the required "type", because it does not support the required operation. In Emerald, we wish to detect such errors earlier. Because Emerald is typed, these errors will often be detected through static type checking at compile time rather than at run time. However, due to the flexibility demanded by our environment, we cannot do complete type checking at compile time, and some type errors will be detectable only at run time. Even when this is the case, type errors will be detected when a type-incorrect *assignment* is attempted rather than when invoking an unimplemented operation on an object. Also, rather than generating error messages that indicate that an operation was not implemented, we wish to generate messages that give a better indication of the reason for the operation not being understood. Making Emerald a typed language allows us to do this, generating messages that indicate that "object O is not of type T."

Second, we feel that type checking can reduce the cost of invocation. Since there is no possibility of "message not understood" errors at invocation time, it follows that no check is necessary. In addition, we have developed an alternative procedure for locating the code to execute in response to an invocation request (doing *method lookup* in Smalltalk terminology) that may provide better performance than existing schemes. This is made possible by the introduction of typing into the language and is described more fully in Section IV-F.

B. Abstract Types

Emerald was designed to be used in the construction of open systems, i.e., those where system-level objects may be created and added to a running system after the basic system is operational. This implies that the type system must be flexible enough to enable old code to invoke newly implemented objects, provided that these objects behave in the expected way. This flexibility and extensibility is provided by both Smalltalk and Eden, but neither of these systems are typed, as was noted above. Emerald wishes to retain this flexibility, but within the framework of a typed programming language.

In order to do this, Emerald objects are typed abstractly. An abstract type defines the interface of an object—the set of operations supported, their *signatures*, and (in principle) their semantics. An operation signature includes the operation name and the number, names, and abstract types of the arguments and results. Each object implementation defines a similar set of operations, but in addition provides 1) a concrete representation for the object and 2) code to implement each of the operations. Abstract types are themselves objects that export a *getSignature* operation. For details on this aspect of the type system, see [4].

The relationship between abstract types and object implementations is many-to-one in both directions: each object may implement several abstract types, and each abstract type may be implemented by several different objects. Fig. 2 illustrates these relationships. The object *DiskFile* implements the abstract type *InputOutputFile*, the abstract types *InputFile* and *OutputFile* (which require only a subset of the *InputOutputFile* operations), and also the abstract type *Any* (which requires no operations at all). The abstract type *InputOutputFile* illustrates that an abstract type may have several implementations, perhaps tuned to different usage patterns. Temporary files may be implemented in primary memory (using *InCoreFile* objects) to provide fast access while giving up permanence in the face of crashes. On the other hand, permanent files implemented using *DiskFiles* would continue to exist across crashes.

This separation of specification and implementation provides the flexibility found in untyped languages such as Smalltalk and Eden, but within the framework of a strongly-typed language. We expect to take advantage of this flexibility in two ways.

- New objects satisfying old interfaces can be added at any time without changing existing objects. For example, consider a windowing system where *Window* is an abstract type. New windows with new functionality can be added, and the window manager can manipulate them without requiring any modifications, or even recompilation or relinking.
- An abstract type may have multiple implementations tailored to particular usage patterns, such as the file example of the previous section. In addition to multiple implementations explicitly provided by the programmer, the compiler itself may generate multiple implementations of

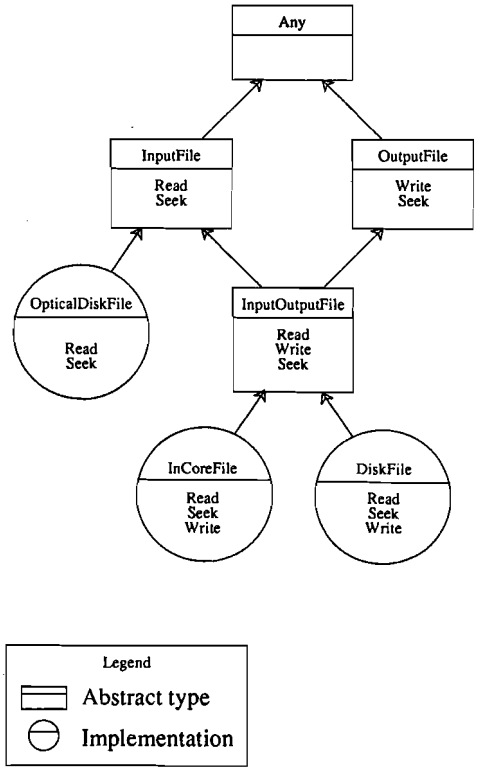

Fig. 2. Abstract types and object implementations.

some objects from the same source code. For example, one implementation might be appropriate for a remote object, another for the case in which the object and its invoker are known to be on the same machine, and yet another when the object is known to be private to some containing object.

An additional benefit of the separation of abstract types and implementations is that it is exactly the separation of specification and implementation necessary for top-down program design and separate compilation. In some existing languages, additional syntax and semantics are required to make possible this separation of specification and implementation. These include external declarations in Concurrent Euclid, Pascal, and Modula, and separate specification and implementation parts in Ada®. The type system of Emerald directly supports separation of specification and implementation; no additional language features are necessary.

C. The Emerald Type System

In Emerald, all identifiers are typed abstractly, i.e., the programmer declares the *abstract* type of the objects that an identifier may name. Such a declaration captures his knowledge of the set of invocations to which those objects should respond.

The notion of type *conformity* is central to Emerald. The legality of an assignment is based on the conformity of the assigned object and the abstract type declared for the identifier. This conformity will always be checked at compile time, except where a run-time check is explicitly requested by the programmer. Roughly, a type P conforms to another type Q if P provides at least the operations of Q. (P may also provide additional operations.) Moreover, the types of the results of P's operations must conform to the types of the results of the corresponding operations of Q. Finally, the types of the arguments of the corresponding operations must conform *in the opposite direction*, i.e., the arguments of Q's operations must conform to those of P's operations.

To illustrate the need for the parameter matching rules, consider the following examples. *Any* is the abstract type containing no operations, thus every type conforms to it.

```
const IntegerPusher ==
  type
    operation Push[Integer]
  end

const AnyPusher ==
  type
    operation Push[Any]
  end
```

These rather useless types define "write-only storage" into which integers and arbitrary objects can be *Pushed*. Intuitively, one would expect *AnyPusher* to conform to *IntegerPusher*, because an implementation of *AnyPusher* can be used wherever an *IntegerPusher* is required. The rules bear this out; the two types are identical except for the argument types of *Push*, and these conform in the opposite direction, i.e., *Integer* conforms to *Any*.

Now consider

```
const IntegerPopper ==
  type
    operation Pop → [Integer]
  end

const AnyPopper ==
  type
    operation Pop → [Any]
  end
```

Here *IntegerPopper* conforms to *AnyPopper*, because the results of *Pop* conform in the *same* direction. Finally, observe that

```
const IntegerStack ==
  type
    operation Push[Integer]
    operation Pop → [Integer]
  end
```

and

```
const AnyStack ==
  type
    operation Push[Any]
    operation Pop → [Any]
  end
```

are incomparable; they do not conform in either direction.

®Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

The reason for this should be obvious; users of an *IntegerStack* object expect its *Pop* operation to return an integer, so an *AnyStack* clearly will not do. Users of an *AnyStack* expect to be able to *Push* arbitrary objects; the *Push* of *IntegerStack* can be applied only to integers.

Emerald type conformity is similar to type compatibility in Owl [30], but differs in a fundamental way from inheritance in Smalltalk. In Smalltalk, a subclass does not necessarily conform to its superclass; for example, it may override some of the operations of the superclass so that they expect different classes of argument. Moreover, one class may conform to another without a subclass relationship existing between them. What a subclass and its superclass *do* have in common is part of their representation and some of their methods. In short, inheritance is a relationship between implementations, while conformity is a relationship between interfaces.

D. Formal Definition of Conformity

The above explanation of conformity in Emerald was not well-founded; the conformity of two types depended on the conformity of the types of the arguments and results of the operations defined by the types. This section will present a formal definition of the conformity relation between types, which forms the basis of Emerald's type system, as well as an algorithm for checking conformity. This presentation of conformity has been simplified in two ways. First, we have omitted operations which are functions and types which are immutable; these would only clutter the presentation with unnecessary detail. Second, we have ignored Emerald's support for polymorphism which is beyond the scope of this paper.

Let T be the set of type names, and let a_i, $r_j \in T$, for $0 \le i \le m$, $0 \le j \le n$. A signature is either an expression of the form

$$a_1, \cdots, a_m \rightarrow r_1, \cdots, r_n$$

which has *arity* $<m, n>$, or the distinguished *null signature*, Λ, on which arity is undefined. Let S be the set of all signatures, and F be the set of operation names. A *type declaration* of $t \in T$ is a total function (also denoted t) from F to S. Obviously, in actually declaring a type, one need only specify the operations which have non-null signatures. For example, the type IntegerStack can be defined as:

$$IntegerStack(x) \equiv \begin{cases} Integer \rightarrow & \text{if } x = Push \\ \rightarrow Integer & \text{if } x = Pop \\ \Lambda & \text{otherwise.} \end{cases}$$

Suppose Θ is a binary relation on T, i.e., $\Theta \subset T \times T$. Intuitively, Θ is a set of pairs which we hope are true assertions about conformity, i.e., $<t, u> \in \Theta$ implies that t conforms to u. However, we have not defined conformity yet, so the intuition cannot be formalized. Now suppose that s, s' are elements of S. Then Θ induces a relation Σ on S by the following definition.

$<s, s'> \in \Sigma$ iff:

i) $s' = \Lambda$, or

ii) arity(s) = arity(s') and writing

$$s = a_1, \cdots, a_m \rightarrow r_1, \cdots, r_n$$

$$s' = a'_1, \cdots, a'_m \rightarrow r'_1, \cdots, r'_n$$

we have

$$<a'_i, a_i> \in \Theta, \quad \text{for } i = 1, 2, \cdots, m$$

$$<r_j, r'_j> \in \Theta, \quad \text{for } j = 1, 2, \cdots, n.$$

Informally, corresponding pairs of results must be in Θ and corresponding pairs of arguments must be in Θ^{-1}. Looking at the conformity of IntegerPopper and AnyPopper from the previous section, we have

$$s = \rightarrow Integer$$

and

$$s' = \rightarrow Any$$

In order for $<s, s'>$ to be in Σ, $<Integer, Any>$ must be in Θ.

Now, what is to distinguish an arbitrary Θ from our desired conformity relation? Exactly the requirement that the corresponding pairs of signatures are in Σ. Formally, we say that Θ is *valid* if, for all type names t and u, and all operation names $f \in F$

$$<t, u> \in \Theta \Rightarrow <t(f), u(f)> \in \Sigma.$$

Lemma: The union of two valid relations is valid.

Proof: Follows immediately from the definitions.

Corollary: There is a unique maximal valid relation, which we will denote by $\gg$.

Finally we have defined conformance. We write $<t, u> \in \gg$ as $t \gg u$ or t conforms to u. One nice thing about valid relations all being contained in $\gg$ is that we know that if two separately defined systems of declarations are each valid, then we will not get any surprises when we combine the declarations. Our concept of conformity is similar to the notion of implicit conversion of types as studied by Reynolds [28], [29].

Now we can define a *decision procedure* that will check whether a given statement of the form "$t \gg u$" is true. Starting from the pair $<t, u>$, we will build up the two relations Θ and Σ recursively. Θ will be a valid relation on T and Σ will be the relation on S induced by Θ. We will do this by, whenever we insert $<a, b>$ into Θ, inserting pairs $<a(f), b(f)>$ into Σ for all f such that $b(f) \neq \Lambda$. This ensures that Θ remains valid. Additionally, whenever we insert $<s, s'>$ into Σ, we insert all the appropriate $<a'_i, a_i>$'s and $<r_j, r'_j>$'s into Θ so that Σ is indeed the derived relation for Θ. But we *fail* in attempting to insert a pair $<a(f), b(f)>$ into Σ if the arities of $a(f)$ and $b(f)$ mismatch, or if $a(f) = \Lambda$ when $b(f) \neq \Lambda$. If we succeed, we will have constructed a valid relation. By the unique maximality of $\gg$, this will prove that $t \gg u$. On the other hand, we only inserted necessary elements into the relations Θ and Σ, so if the procedure *fails*, then t does not conform to u.

Let us apply this decision procedure to check whether IntegerStack ⊳ AnyStack. In order to insert <*IntegerStack, AnyStack*> into Θ, we must insert

$$<\text{IntegerStack(Pop)}, \text{AnyStack(Pop)}>$$
$$= <\rightarrow \text{Integer}, \rightarrow \text{Any}>$$

and

$$<\textit{IntegerStack(Push)}, \textit{AnyStack(Push)}>$$
$$= <\textit{Integer}\rightarrow, \textit{Any}\rightarrow>$$

into Σ. Inserting $<\rightarrow \textit{Integer}, \rightarrow \textit{Any}>$ into Σ causes us to insert <*Integer, Any*> into Θ which causes no further insertions in Σ since $\textit{Any}(x) = \Lambda$ for all x. Attempting to insert $<\textit{Integer}\rightarrow, \textit{Any}\rightarrow>$ into Σ causes us to insert <*Any, Integer*> into Θ, which in turn causes us to attempt to insert $<\textit{Any}(+), \textit{Integer}(+)>$ into Σ which fails since $\textit{Any}(+) = \Lambda$ and $\textit{Integer}(+) \neq \Lambda$. Thus we conclude that IntegerStack does not conform to AnyStack.

Note that there is no need to start with empty relations Θ and Σ; any valid relation Θ on types and its induced relation Σ on signatures may be used as a starting point. In actually implementing this procedure, the relations Θ and Σ may be retained after conformity checking, thus eliminating the need to recompute them.

E. Comparison to Other Languages

One currently popular view of a data type is as a set of operations. That is, each data type is a set of operations that may be applied to values of the type, thereby providing an interpretation for those values [11], [23]. There are, however, two approaches to the question of the ownership of these operations.

A number of languages, including Russell [9] and CLU [23], consider the operations to be *fields* of the type; that is, the operations are owned by the type, and are selected from the type. As an example, consider the type *Integer*. In Russell or CLU, the type would own operations like:

operation 0 → *Integer*
operation + (*Integer* × *Integer*) → *Integer*
operation * (*Integer* × *Integer*) → *Integer*

In order to add two integers in CLU (Russell is similar), one selects the + operation from the Integer type to apply to the values:

result := *Integer*$+(*argument1*, *argument2*)

The other approach is exemplified by Smalltalk. In Smalltalk, the operations that may manipulate objects are broken into two disjoint sets. Some operations, generally those that create new objects, are defined as messages to (operations on) a distinguished object called the *class*. An example of a class operation is creating an integer value from a character string representing the value. The other set of operations are defined as messages to the object itself; the operations + and * on integers are examples. The Smalltalk statement that adds two integers and assigns the result to a variable is:

result ← *target* + *argument*

This statement is interpreted to mean "send the message with the name + to the object *target*, providing the object *argument* as argument, and assign the resulting object to the variable *result*."

In Emerald, we have adopted an approach similar to Smalltalk's. Our primary motivation was a concern over distribution. In the CLU and Russell approach, the statements in the body of an operation have access to the representation of an arbitrary number of objects of the type of interest. In particular, the implementor of the addition operation on integers is allowed to access the bits that make up the representation of *both* of the arguments. In the face of distribution, implementing this is potentially very complicated. Either the two objects need to be moved to a common location in order to perform the operation, or else the implementation needs to be able to perform arbitrary primitive operations remotely. In our view, only the object that is performing the operation has its representation accessible, and all other objects, including others of the same type, must be accessed through invocations.

The view of operations as being owned by objects is also required by our separation of abstract types and implementations. Since the two integers being added may in fact be implemented differently, the only way of implementing the integer operations is by invoking lower level operations that each concrete type may implement differently.

F. Type Implementation Considerations

A major consequence of the flexibility of Emerald's type system is that multiple objects are allowed to implement a single abstract type. In principle, multiple implementations of a data type cause no problems because operations are invoked by name, and it is up to the invoked object to understand the name. In practice, it is important to optimize this process.

In Smalltalk and Eden, the cost of invocation by name appears at *invocation time* since the appropriate code for each invocation must be located when the invocation is attempted. Substantial effort has been put into implementations of Smalltalk that reduce the overhead of code location (*method lookup* in Smalltalk terminology) to an acceptable level [10], [21]. Techniques include caching of operation addresses at call sites and making likely guesses about the implementations of objects with certain abstract types (like Integer), thereby optimizing for the common case. Since our object model is similar to that of Smalltalk, all of the optimizations that can be done in the implementation of Smalltalk could also be done in Emerald. In addition, the fact that Emerald is statically typed allows an alternative code location strategy that may yield better performance.

With each identifier, we associate a vector of operations. The length of this vector is determined by the identifier's abstract type, since only the operations defined by

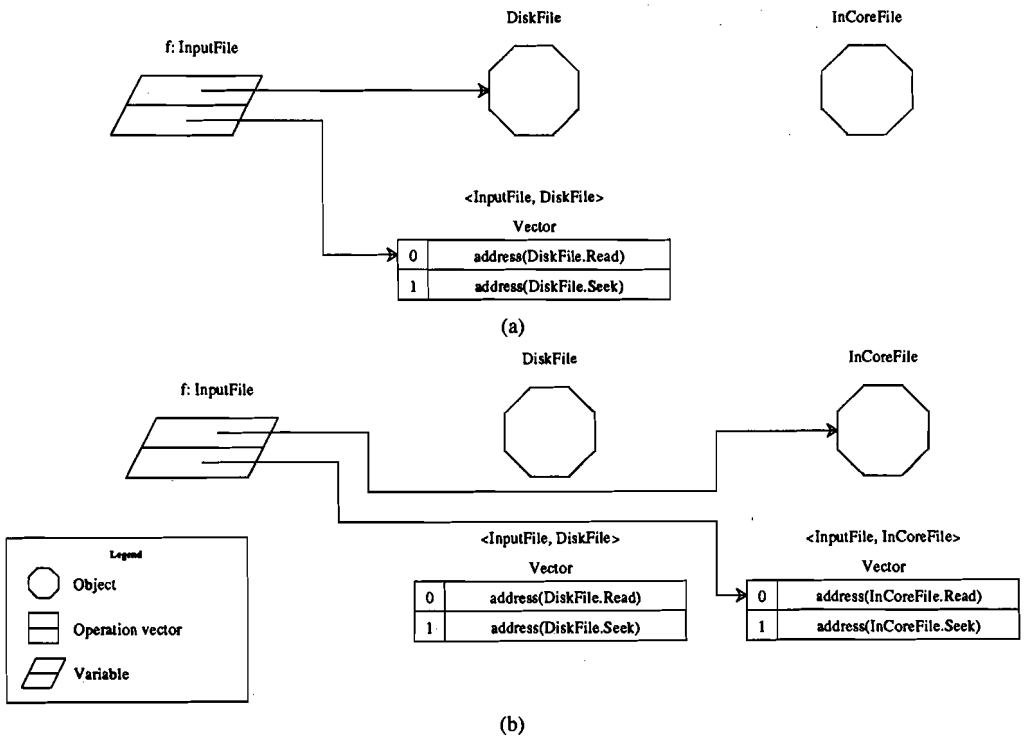

Fig. 3. (a) Before the assignment. (b) After the assignment.

the abstract type may be invoked on the object named by the identifier. The content of the vector is determined by the implementation of the object currently named by the identifier, since it consists of the addresses of the machine code sequences that implement the operations. Consider a variable f of abstract type a. The operations that may be performed on f are determined by a; the addresses of the appropriate code for these operations depend on the implementation c of the object named by f. Thus the pair $<a, c>$ uniquely determines an *operation vector* associated with f. In Fig. 3(a), a is *InputFile* and c is *DiskFile*. The vector has one element for each of the *InputFile* operations *read* and *seek*; the values of these elements are the addresses of the corresponding *DiskFile* routines.

When an assignment is made to f, the contents of its operation vector may need to be changed. For example, when an *InCoreFile* object is assigned to f, the operation vector associated with f must become appropriate to the pair $<$*InputFile*, *InCoreFile*$>$, as shown in Fig. 3(b).

This scheme replaces the method lookup required by Smalltalk by a single indexing operation. The cost is an additional word of storage for the pointer to the operation vector, and occasional recomputation of the elements of these vectors on assignment. The operation vectors themselves may be shared between all identifiers of identical abstract type that name objects with the same implementation, since it is the pair <abstract type, implementation> that determines the contents of the vector.

V. Distribution Support

As previously stated, the principal objective of Emerald is to simplify the construction of distributed programs. System concepts such as concurrency, multiple nodes, and object location are integrated into the language. This differs from, for example, EPL, where distribution is layered on an existing language through the use of a preprocessor, and from Accent [27], where distribution is provided as an operating system facility.

In Emerald, objects encapsulate the notions of process and data and are the natural unit of distribution. At any time each Emerald object is located at a specific node. Conceptually, a node is an object of a system-defined type. Node objects support node-specific operations, thereby allowing objects to invoke kernel operations. Such access to the underlying kernel is analogous to that provided by *kernel ports* in Accent.

Programmers may choose to ignore or exploit the concept of object location. In a distributed system, objects must be able to invoke other objects in a location-independent manner. This facility makes network services transparently accessible. In Emerald, locating the target of an invocation is the responsibility of the system. An object is permitted to move between successive invocations, or even during an invocation. While applications can control the placement of objects, most applications can ignore location considerations since the semantics of local and remote invocation are identical.

Nevertheless, there are two reasons for making location visible to the programmer: performance and availability. In a network, the efficiency of interobject communication is obviously a function of location. An application can colocate objects that communicate intensely and thus reduce the communication overhead. Alternatively, numerical applications can achieve significant performance gains by placing concurrent subcomputations on different nodes. An object manager may increase availability by placing replicas of its objects on different nodes.