This empty page was substituted for a blank page in the original document.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Overview</td>
<td>3</td>
</tr>
<tr>
<td>1.1. Objects and Variables</td>
<td>3</td>
</tr>
<tr>
<td>1.2. Assignment and Calls</td>
<td>4</td>
</tr>
<tr>
<td>1.3. Type Correctness</td>
<td>4</td>
</tr>
<tr>
<td>1.4. Rules and Guidelines</td>
<td>4</td>
</tr>
<tr>
<td>1.5. Program Structure</td>
<td>5</td>
</tr>
<tr>
<td>2. Concepts for Distributed Programs</td>
<td>7</td>
</tr>
<tr>
<td>2.1. Guardians</td>
<td>7</td>
</tr>
<tr>
<td>2.2. Actions</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1. Nested Actions</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2. Atomic Objects and Atomic Types</td>
<td>9</td>
</tr>
<tr>
<td>2.2.3. Nested Topactions</td>
<td>11</td>
</tr>
<tr>
<td>2.3. Remote Calls</td>
<td>11</td>
</tr>
<tr>
<td>2.4. Transmittable Types</td>
<td>12</td>
</tr>
<tr>
<td>2.5. Orphans</td>
<td>12</td>
</tr>
<tr>
<td>2.6. Deadlocks</td>
<td>13</td>
</tr>
<tr>
<td>3. Environment</td>
<td>15</td>
</tr>
<tr>
<td>3.1. The Library</td>
<td>15</td>
</tr>
<tr>
<td>3.2. Independence of Guardian Images</td>
<td>15</td>
</tr>
<tr>
<td>3.3. Guardian Creation</td>
<td>15</td>
</tr>
<tr>
<td>3.4. The Catalog</td>
<td>15</td>
</tr>
<tr>
<td>4. Notation</td>
<td>17</td>
</tr>
<tr>
<td>5. Lexical Considerations</td>
<td>19</td>
</tr>
<tr>
<td>5.1. Reserved Words</td>
<td>19</td>
</tr>
<tr>
<td>5.2. Identifiers</td>
<td>19</td>
</tr>
<tr>
<td>5.3. Literals</td>
<td>20</td>
</tr>
<tr>
<td>5.4. Operators and Punctuation Tokens</td>
<td>20</td>
</tr>
<tr>
<td>5.5. Comments and Other Separators</td>
<td>20</td>
</tr>
<tr>
<td>6. Types, Type Generators, and Type Specifications</td>
<td>21</td>
</tr>
<tr>
<td>6.1. Type inclusion</td>
<td>22</td>
</tr>
<tr>
<td>6.2. The Sequential Built-In Types and Type-generators</td>
<td>22</td>
</tr>
<tr>
<td>6.2.1. Null</td>
<td>22</td>
</tr>
<tr>
<td>6.2.2. Bool</td>
<td>22</td>
</tr>
<tr>
<td>6.2.3. Int</td>
<td>22</td>
</tr>
<tr>
<td>6.2.4. Real</td>
<td>23</td>
</tr>
<tr>
<td>6.2.5. Char</td>
<td>23</td>
</tr>
<tr>
<td>6.2.6. String</td>
<td>24</td>
</tr>
<tr>
<td>6.2.7. Any</td>
<td>24</td>
</tr>
<tr>
<td>6.2.8. Sequence Types</td>
<td>24</td>
</tr>
<tr>
<td>6.2.9. Array Types</td>
<td>25</td>
</tr>
<tr>
<td>6.2.10. Structure Types</td>
<td>25</td>
</tr>
<tr>
<td>6.2.11. Record Types</td>
<td>26</td>
</tr>
<tr>
<td>6.2.12. Oneof Types</td>
<td>27</td>
</tr>
<tr>
<td>6.2.13. Variant Typses</td>
<td>28</td>
</tr>
<tr>
<td>6.2.14. Procedure and Iterator Types</td>
<td>29</td>
</tr>
<tr>
<td>6.3. Atomic_Array, Atomic_Record, and Atomic_Variant</td>
<td>30</td>
</tr>
<tr>
<td>6.4. Guardian Types</td>
<td>31</td>
</tr>
<tr>
<td>6.5. Handler and Creator Types</td>
<td>32</td>
</tr>
</tbody>
</table>
Table of Contents

6.6. Image 32
6.7. Mutex 33
6.8. Node 34
6.9. Other Type Specifications 34

7. Scopes, Declarations, and Equates 35
   7.1. Scoping Units 35
      7.1.1. Variables 36
      7.1.2. Declarations 36
   7.2. Equates and Constants 37
      7.2.1. Abbreviations for Types 38
      7.2.2. Constant Expressions 38

8. Assignment and Calls 39
   8.1. Assignment 39
      8.1.1. Simple Assignment 39
      8.1.2. Multiple Assignment 39
   8.2. Local Calls 40
   8.3. Handler Calls 41
      8.3.1. Semantics of Handler Calls 43
   8.4. Creator Calls 44
      8.4.1. Semantics of Creator Calls 44

9. Expressions 47
   9.1. Literals 47
   9.2. Variables 47
   9.3. Parameters 47
   9.4. Equated Identifiers 47
   9.5. Equate Module References 47
   9.6. Self 48
   9.7. Procedure, Iterator, and Creator Names 48
   9.8. Blind 48
   9.9. Procedure Calls 50
   9.10. Handler Calls 50
   9.11. Creator Calls 51
   9.12. Selection Operations 51
      9.12.1. Element Selection 51
      9.12.2. Component Selection 51
   9.13. Constructors 52
      9.13.1. Sequence Constructors 52
      9.13.2. Array and Atomic Array Constructors 52
      9.13.3. Structure, Record, and Atomic Record Constructors 52
   9.15. Cand and Cor 54
   9.16. Precedence 54
   9.17. Up and Down 55

10. Statements 57
    10.1. Calls 57
    10.2. Update Statements 58
        10.2.1. Element Update 58
        10.2.2. Component Update 58
    10.3. Block Statement 58
    10.4. Fork Statement 58
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.5. Commuting Operations</td>
<td>102</td>
</tr>
<tr>
<td>15.6. Multiple Mutexes</td>
<td>104</td>
</tr>
<tr>
<td>Appendix I. Syntax</td>
<td>107</td>
</tr>
<tr>
<td>Appendix II. Built-in Types and Type Generators</td>
<td>119</td>
</tr>
<tr>
<td>II.1. Null</td>
<td>120</td>
</tr>
<tr>
<td>II.2. Nodes</td>
<td>120</td>
</tr>
<tr>
<td>II.3. Booleans</td>
<td>121</td>
</tr>
<tr>
<td>II.4. Integers</td>
<td>121</td>
</tr>
<tr>
<td>II.5. Reals</td>
<td>123</td>
</tr>
<tr>
<td>II.6. Characters</td>
<td>125</td>
</tr>
<tr>
<td>II.7. Strings</td>
<td>126</td>
</tr>
<tr>
<td>II.8. Sequences</td>
<td>128</td>
</tr>
<tr>
<td>II.9. Arrays</td>
<td>130</td>
</tr>
<tr>
<td>II.10. Atomic Arrays</td>
<td>133</td>
</tr>
<tr>
<td>II.11. Structs</td>
<td>138</td>
</tr>
<tr>
<td>II.12. Records</td>
<td>139</td>
</tr>
<tr>
<td>II.13. Atomic Records</td>
<td>141</td>
</tr>
<tr>
<td>II.14. Oneofs</td>
<td>143</td>
</tr>
<tr>
<td>II.15. Variants</td>
<td>144</td>
</tr>
<tr>
<td>II.16. Atomic Variants</td>
<td>146</td>
</tr>
<tr>
<td>II.17. Procedures and Iterators</td>
<td>148</td>
</tr>
<tr>
<td>II.18. Handlers and Creators</td>
<td>149</td>
</tr>
<tr>
<td>II.19. Anyes</td>
<td>150</td>
</tr>
<tr>
<td>II.20. Images</td>
<td>150</td>
</tr>
<tr>
<td>II.21. Mutexes</td>
<td>151</td>
</tr>
<tr>
<td>Appendix III. Rules and Guidelines for Using Argus</td>
<td>153</td>
</tr>
<tr>
<td>III.1. Serializability and Actions</td>
<td>153</td>
</tr>
<tr>
<td>III.2. Actions and Exceptions</td>
<td>153</td>
</tr>
<tr>
<td>III.3. Stable Variables</td>
<td>154</td>
</tr>
<tr>
<td>III.4. Transmission and Transmissability</td>
<td>154</td>
</tr>
<tr>
<td>III.5. Mutex</td>
<td>154</td>
</tr>
<tr>
<td>III.6. User-Defined Atomic Objects</td>
<td>156</td>
</tr>
<tr>
<td>III.7. Subordinate Where Clauses</td>
<td>157</td>
</tr>
<tr>
<td>Appendix IV. Changes from CLU</td>
<td>159</td>
</tr>
<tr>
<td>IV.1. Exception Handling</td>
<td>159</td>
</tr>
<tr>
<td>IV.2. Type Any</td>
<td>159</td>
</tr>
<tr>
<td>IV.3. Built-In Types</td>
<td>159</td>
</tr>
<tr>
<td>IV.4. Type Inclusion</td>
<td>160</td>
</tr>
<tr>
<td>IV.5. Where Clauses</td>
<td>160</td>
</tr>
<tr>
<td>IV.6. Uninitialized Variables</td>
<td>160</td>
</tr>
<tr>
<td>IV.7. Lexical Changes</td>
<td>160</td>
</tr>
<tr>
<td>IV.8. Input/Output Changes</td>
<td>160</td>
</tr>
<tr>
<td>Index</td>
<td>161</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2-1: Locking and Version Management Rules for a Subaction S, on Object X 10
Figure 13-1: Spooler Guardian 91
Figure 14-1: Partial implementation of table. 95
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-1</td>
<td>Reserved Words</td>
<td>19</td>
</tr>
<tr>
<td>6-2</td>
<td>Operators and Precedence Notation</td>
<td>20</td>
</tr>
<tr>
<td>6-1</td>
<td>Statements and Control Structures</td>
<td>24</td>
</tr>
<tr>
<td>6-1</td>
<td>Rules and Expressions</td>
<td>33</td>
</tr>
<tr>
<td>9-2</td>
<td>Functions and Instructions</td>
<td>54</td>
</tr>
<tr>
<td>10-1</td>
<td>Language Elements</td>
<td>68</td>
</tr>
<tr>
<td>5-1</td>
<td>Common Language Expressions</td>
<td>110</td>
</tr>
</tbody>
</table>
This empty page was substituted for a blank page in the original document.
Guide to the Manual

This document serves both as a reference manual and as an introduction to Argus. Sections 1 through 3 present an overview of the language. These sections highlight the essential features of Argus. Sections 4 through 15 and the appendices form the reference manual proper. These sections describe each aspect of Argus in detail, and discuss the proper use of various features. Appendices I and II provide summaries of Argus's syntax and data types. Appendix III summarizes some of the pragmatic rules for using Argus.

Since Argus is based on the programming language CLU, the reader is expected to have some familiarity with CLU. Those readers needing an introduction to CLU might read Liskov, B. and Guttag, J., Abstraction and Specification in Program Development (MIT Press, Cambridge, 1986). A shorter overview of CLU appears in the article Liskov, B., et al., "Abstraction Mechanisms in CLU" (Comm. ACM, volume 20, number 8 (Aug. 1977), pages 564-576). Appendix IV summarizes the changes made to Argus that are not upward compatible with CLU.

An overview and rationale for Argus is presented in Liskov, B. and Scheifler, R., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs" (ACM Transactions on Programming Languages and Systems, volume 5, number 3 (July 1983), pages 381-404).

The Preliminary Argus Reference Manual appeared as Programming Methodology Group Memo 39 in October 1983. Since that time several new features have been added to the language; the most significant of these are closures (see Section 9.8), a fork statement (see Section 10.4), equate modules (see Section 12.4), and a more flexible instantiation mechanism (see Section 12.6). An earlier version of this document appeared as Programming Methodology Group Memo 54 in March 1987; this version is essentially identical, except that the locking policy for the built-in type generator atomic_array has been simplified.

We would greatly appreciate receiving comments on both the language and this manual. Comments should be sent to: Professor Barbara Liskov, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

The authors thank all the members of the Programming Methodology group at MIT for their help and suggestions regarding the language and this manual, with special thanks going to Elliot Kodelner, Deborah Hwang, Sharon Perl, and the authors of the CLU Reference Manual.
Though her unhappy rival was hers to keep
Queen Juno also had a troubled mind:
What would Jove turn to next? Better, she thought,
To give the creature to Aeaeus's son,
The frightful Argus whose unnatural head
Shone with a hundred eyes, a perfect jailer
For man or beast: the hundred eyes took turns
At staring wide awake in pairs, and two
At falling off to sleep; no matter how or
Where he stood he gazed at Io; even when
His back was turned, he held his prisoner
In sight and in his care.

—Ovid, *The Metamorphoses*, Book 1
Translated by H. Gregory
The Viking Press, Inc., New York, 1958
1. Overview

Argus is an experimental language/system designed to support the construction and execution of distributed programs. Argus is intended to support only a subset of the applications that could benefit from being implemented by a distributed program. The programs designed these applications: they make use of on-line data that must remain accessible during system or hardware failures, and they provide services under real-time constraints (see see Section 13.1). Examples of such applications are office automation systems and faulting systems.

Argus is based on CLU. It is largely an extension of CLU, but there are number of differences (see Appendix IV). Like CLU, Argus provides programming for procedure abstraction, handles for control abstraction, and clusters for data abstraction. It also provides handles for procedures that correspond to control access to one or more resources. These and other differences are described in Section 2. Argus also provides equate modules as a convenient way to refer to procedures (see Section 13.4). As in CLU, modules may be parameterized, so that a single module can fulfill a role or related obligations.

1.1. Objects and Variables

The semantics of Argus deal with objects and variables. Objects are the data entities that are created and manipulated by applications. Variables are the names used to refer to objects.

Every object has a type that characterizes its behavior. A type defines a set of primitive operations to create and manipulate objects of that type.

An object may refer to other objects or even to itself. It is also possible for an object to be referred to or shared by several objects. Objects exist independently of processes and message transmissions.

There are several categories of objects in Argus. An object with a unique, time-varying behavior is called a mutable object. A mutable object has data that may be updated under some circumstances without changing the object's identity. A mutable object can have both global and cluster handles (see Section 2.2.1). An object whose state is inseparable from its identity. An immutable object has a cluster handles (see Section 2.2.2). Objects are immutable if they can be used as arguments to calls of different procedures (see Section 2.4). Since guardians, handlers, and variables are functions, the shared among procedures, these objects are said to be global objects. All other variables, such as arguments, integers, or procedures, can only be shared within a single guardian and the associated objects.

Variables are names used in programs to denote specific objects at execution time. It is possible for two variables to denote the same object. Variables are not objects, they cannot be shared by other variables or referred to by objects.

Variables in guardian modules can be declared to be stable. The objects denoted by stable variables survive crashes (see Section 2) and are called stable objects.
1.2. Assignment and Calls

The basic events in Argus are assignments and calls. The assignment statement \( x := E \), where \( x \) is a variable and \( E \) is an expression, causes \( x \) to denote the object resulting from the evaluation of \( E \). The object is not copied.

A call involves passing argument objects from the caller to the called routine and returning result objects from the routine to the caller. For local calls, argument passing is defined in terms of assignment, or call by sharing; for remote calls, call by value is used. In a local call, the formal arguments of a routine are considered to be local variables of the routine and are initialized, by assignment, to the objects resulting from the evaluation of the argument expressions. In a remote call (see Section 2.3), a copy of the objects resulting from the evaluation of the argument expressions is made and transmitted to the called handler or creator (see Section 2.4). These copies are then used to initialize the formal arguments as before. Local objects are shared between the caller and a called procedure or iterator, but local objects are never shared between the caller and a called handler or creator.

1.3. Type Correctness

The declaration of a variable specifies the type of the objects which the variable may denote. In a legal assignment statement, \( x := E \), the type of the expression \( E \) must be included in the type of the variable \( x \). Type inclusion is essentially equality of types (see Section 12.6), except for routine types. (A routine type with fewer exceptions is included in an otherwise identical routine type with more exceptions. See Section 6.1 for details.)

Argus is a type-safe language, in that it is not possible to treat an object of type \( T \) as if it were an object of some other type \( S \) (the one exception is when \( T \) is a routine type and \( S \) includes \( T \)). The type safety of Argus, plus the restriction that only the code in a cluster may convert between the abstract type and the concrete representation (see Section 12.3), ensure that the behavior of an object can be characterized completely by the operations of its type.

1.4. Rules and Guidelines

Throughout this manual, and especially in the discussions of atomicity, there are pragmatic rules and guidelines for the use of the language. Certain properties that the language would like to guarantee, for example that atomic actions are really atomic, are difficult or impossible for the language to guarantee completely. As in any useful programming language, programmers have enough rope to hang themselves. The rules and guidelines noted throughout the manual (and collected in Appendix III) try to make the responsibilities of the language and the programmer clear.
1.5. Program Structure

An Argus distributed application consists of one or more guardians, defined by guardian modules. Guardian modules may in turn use all the other kinds of modules that Argus provides. Argus programmers may also write single-machine programs with no stable state, using Argus as essentially a "concurrent CLU." Such programs may be used to start up multi-guardian applications. Each module is a separate textual unit, and is compiled independently of other modules. Compilation is discussed in Section 3.
2. Concepts for Distributed Programs

In this chapter we present an overview of the new concepts in Argus that support distributed programs. In Section 2.1, we discuss guardians, the module used in Argus to distribute data. Next, in Section 2.2, we present atomic actions, which are used to cope with concurrency and failure. In Section 2.3 we describe remote calls, the inter-guardian communication mechanism. In Section 2.4 we discuss transmissible types: types whose objects can be sent as arguments or results of remote calls. Finally, in Section 2.4 we discuss orphans.

2.1. Guardians

Distributed applications are implemented in Argus by one or more modules called guardians. A guardian abstraction is a kind of data abstraction, but it differs from the data abstractions supported by clusters (as found in CLU). In general, data abstractions consist of a set of operations and a set of objects. In a cluster the operations are considered to belong to the abstraction as a whole. However, guardian instances are objects and their handlers are their operations. Guardian abstraction is similar to the data abstractions in Simula and Smalltalk-80; guardians are like class instances.

A node is a single physical location, which may have multiple processors. A guardian instance resides at a single node, although a node may support several guardians. A guardian encapsulates and controls access to one or more resources, such as data or devices. Access to the protected resource is provided by a set of operations called handlers. Internally, a guardian consists of a collection of data objects and processes that can be used to manipulate those objects. In general, there will be many processes executing concurrently in a guardian: a new process is created to execute each handler call, processes may be explicitly created, and there may be other processes that carry out background activity of the guardian.

The data objects encapsulated by a guardian are local: they cannot be accessed directly by a process in another guardian. In contrast, guardians are global objects: a single guardian may be shared among processes at several different guardians. A process with a reference to a guardian can call the guardian’s handlers, and these handlers can access the data objects inside the guardian. Handler calls allow access to a guardian’s local data, but the guardian controls how that data can be manipulated.

When a node fails, it crashes. A crash is a "clean" failure, as opposed to a "Byzantine" failure. A guardian survives crashes of its node (with as high a probability as needed). A guardian’s state consists of stable and volatile objects. When a guardian’s node crashes, all processes running inside the guardian at the time of the crash are lost, along with the guardian’s volatile objects, but the guardian’s stable objects survive the crash. Upon recovery of the guardian’s node, the guardian runs a special recovery process to reconstruct its volatile objects from its stable objects. Since the volatile objects are lost in a crash, they typically consist only of redundant data that is used to improve performance (for example, an index into a database). The persistent state of an application should be kept in stable objects.

Guardians are implemented by guardian definitions. These define:
1. The *creators*. These are operations that can be called to create new guardian instances that perform in accordance with the guardian definition.

2. The guardian's stable and volatile state.

3. The guardian's handlers.

4. The *background code*. This is code that the guardian executes independent of any handler calls, for example, to perform some periodic activity.

5. The *recovery code*. This is code that is executed after a crash to restore the volatile objects.

Guardians and guardian definitions are discussed in Section 13.

### 2.2. Actions

The distributed data in an Argus application can be shared by concurrent processes. A process may attempt to examine and transform some objects from their current states to new states, with any number of intermediate state changes. Interactions among concurrent processes can leave data in an inconsistent state. Failures (for example, node crashes) can occur during the execution of a process, raising the additional possibility that data will be left in an inconsistent intermediate state. To support applications that need consistent data, Argus permits the programmer to make processes atomic.

We call an atomic process an *action*. Actions are atomic in that they are both serializable and recoverable. By serializable, we mean that the overall effect of executing multiple concurrent actions is as if they had been executed in some sequential order, even though they actually execute concurrently. By recoverable, we mean that the overall effect of an action is "all-or-nothing:" either all changes made to the data by the action happen, or none of these changes happen. An action that completes all its changes successfully commits; otherwise it aborts, and objects that it modified are restored to their previous states.

Before an action can commit, new states of all modified, stable objects must be written to stable storage\(^1\): storage that survives media crashes with high probability. Argus uses a two-phase commit protocol\(^2\) to ensure that either all of the changes made by an action occur or none of them do. If a crash occurs after an action modifies a stable object, but before the new state has been written to stable storage, the action will be aborted.

#### 2.2.1. Nested Actions

Actions in Argus can be nested: an action may be composed of several *subactions*. Subactions can be used to limit the scope of failures and to introduce concurrency within an action.

An action may contain any number of subactions, some of which may be performed sequentially, some

---


concurrently. This structure cannot be observed from outside the action; the overall action is still atomic. Subactions appear as atomic actions with respect to other subactions of the same parent. Thus, subactions can be executed concurrently.

Subactions can commit and abort independently, and a subaction can abort without forcing its parent action to abort. However, the commit of a subaction is conditional: even if a subaction commits, aborting its parent action will abort it.

The root of a tree of nested actions is called a topaction. Topactions have no parent; they cannot be aborted once they have committed. Since the effects of a subaction can always be undone by aborting its parent, the two-phase commit protocol is used only when topactions attempt to commit.

In Argus, an action (e.g., a handler call) may return objects through either a normal return or an exception and then abort. The following rule should be followed to avoid violating serializability: a subaction that aborts should not return any information obtained from data shared with other concurrent actions.

### 2.2.2. Atomic Objects and Atomic Types

Atomicity of actions is achieved via the data objects shared among those actions. Shared objects must be implemented so that actions using them appear to be atomic. Objects that support atomicity are referred to as atomic objects. Atomic objects provide the synchronization and recovery needed to ensure that actions are atomic. An atomic type is a type whose objects are all atomic. Some objects do not need to be atomic: for example, objects that are local to a single process. Since the synchronization and recovery needed to ensure atomicity may be expensive, we do not require that all types be atomic. (For example, Argus provides all the built-in mutable types of CLU; these types are not atomic.) However, it is important to remember that atomic actions must share only atomic objects.

Argus provides a number of built-in atomic types and type generators. The built-in scalar types (null, node, bool, char, int, real, and string) are atomic. Parameterized types can also be atomic. Typically, an instance of a type generator will be atomic only if any actual type parameters are also atomic. The built-in immutable type generators (sequence, struct, and oneof) are atomic if their parameter types are atomic. In addition, Argus provides three mutable atomic type generators: atomic_array, atomic_record, and atomic_variant. The operations on these types are nearly identical to the normal array, record, and variant types of CLU. Users may also define their own atomic types (see Section 15).

The implementation of the built-in mutable atomic type generators is based on a simple locking model. There are two kinds of locks: read locks and write locks. When an action calls an operation on an atomic object, the implementation acquires a lock on that object in the appropriate mode: it acquires a write lock if it mutates the object, or a read lock if it only examines the object. The built-in types allow multiple concurrent readers, but only a single writer. If necessary, an action is forced to wait until it can obtain the appropriate lock. When a write lock on an object is first obtained by an action, the system makes a copy
of the object’s state in a new version, and the operations called by the action work on this version. If, ultimately, the action commits, this version will be retained, and the old version discarded. A subaction’s locks are given to its parent action when it commits. When a topaction commits, its locks are discarded and its effects become visible to other actions. If the action aborts, the action’s locks and the new version will be discarded, and the old version retained (see Figure 2-1).

Figure 2-1: Locking and Version Management Rules for a Subaction S, on Object X

---

**Acquiring a read lock:**

All holders of write locks on X must be ancestors of S.

**Acquiring a write lock:**

All holders of read and write locks on X must be ancestors of S.

If this is the first time S has acquired a write lock on X, push a copy of X on the top of its version stack.

**Commit:**

S’s parent acquires S’s lock on X.

If S holds a write lock on X, then S’s version becomes S’s parent’s version.

**Abort:**

S’s lock and version (if any) are discarded.

---

More precisely, an action can obtain a read lock on an object if every action holding a write lock on that object is an ancestor of the requesting action. An action can obtain a write lock on an object if every action holding a (read or write) lock on that object is an ancestor. When a subaction commits, its locks are inherited by its parent and its new versions replace those of its parent; when a subaction aborts, its locks and versions are discarded (see Figure 2-1). Because Argus guarantees that parent actions never run concurrently with their children, these rules ensure that concurrent actions never hold write locks on the same object simultaneously.

The ancestors of a subaction are itself, its parent, its parent’s parent, and so on; a subaction is a descendant of its ancestors. A subaction commits to the top if it and all its ancestors, including the topaction, commit. A subaction is a committed descendant of an ancestor action if the subaction and all intervening ancestors have committed. When a topaction attempts to commit, the two-phase commit protocol is used to ensure that the new versions of all objects modified by the action and all its committed descendants are copied to stable storage. After the new versions have been recorded stably, the old versions are thrown away.

User-defined atomic types can provide greater concurrency than built-in atomic types. An

---

3. This operational description (and others in this manual) is not meant to constrain implementors. However, this particular description does reflect our current implementation.

4. An example can be found in Weihl, W. and Liskov, B., "Implementation of Resilient, Atomic Data Types," ACM Transactions on Programming Languages and Systems, volume 7, number 2 (April 1985), pages 244-269.
2.2.2 Atomic Objects and Atomic Types

implementation of a user-defined atomic type must address several issues. First, it must provide proper synchronization so that concurrent calls of its operations do not interfere with each other, and so that the actions that call its operations are serialized. Second, it must provide recovery for actions using its objects so that aborted actions have no effect. Finally, it must ensure that changes made to its objects by actions that commit to the top are recorded properly on stable storage. The built-in atomic types and the mutex type generator are useful in coping with these issues. User-defined atomic types are discussed further in Section 15.

2.2.3. Nested Topactions

In addition to nesting subactions inside other actions, it is sometimes useful to start a new topaction inside another action. Such a nested topaction, unlike a subaction, has no special privileges relative to its "parent"; for example, it is not able to read an atomic object modified by its "parent". Furthermore, the commit of a nested topaction is not relative to its "parent"; its versions are written to stable storage, and its locks are released, just as for normal topactions.

Nested topactions are useful for benevolent side effects that change the representation of an object without affecting its abstract state. For example, in a naming system a name look-up may cause information to be copied from one location to another, to speed up subsequent look-ups of that name. Copying the data within a nested topaction that commits ensures that the changes remain in effect even if the "parent" action aborts.

A nested topaction is used correctly if it is serializable before its "parent". This is true if either the nested topaction performs a benevolent side effect, or if all communication between the nested topaction and its parent is through atomic objects.

2.3. Remote Calls

An action running in one guardian can cause work to be performed at another guardian by calling a handler provided by the latter guardian. An action can cause a new guardian to be created by calling a creator. Handler and creator calls are remote calls. Remote calls are similar to local procedure calls; for example, the calling process waits for the call to return. Remote calls differ from local procedure calls in several ways, however.

First, the arguments and results of a remote call are passed by value (see below and also Section 14) rather than by sharing. This ensures that the local objects of one guardian remain local to that guardian, even if their values are used as arguments or results of remote calls to other guardians. The only objects that are passed by sharing in remote calls are the global objects: guardians, handlers, creators, and nodes.

Second, any remote call can raise the exceptions failure and unavailable. (Unlike CLU, not all local calls can raise failure, see Appendix IV.) The occurrence of failure means that the call is unlikely to ever succeed, so there is no point in retrying the call in the future. Unavailable, on the other hand, means that
the call should succeed if retried in the future, but is unlikely to succeed if retried immediately. For example, failure can arise because it is impossible to transmit the arguments or results of the call (see Section 14); unavailable can arise if the guardian being called has crashed, or if the network is partitioned.

Third, a handler or creator can be called only from inside an action, and the call runs as a subaction of the calling action. This ensures that a remote call succeeds at most once: either a remote call completes successfully and commits, or it aborts and all of its modifications are undone (provided, of course, that the actions involved are truly atomic). Although the effect of a remote call occurs at most once, the system may need to attempt it several times; this is why remote calls are made within actions.

2.4. Transmissible Types

Arguments and results of remote calls are passed by value. This means that the argument and result objects must be copied to produce distinct objects. Not all objects can be copied like this; those that can are called transmissible objects, and their types are called transmissible types. Only transmissible objects may be used as arguments and results of a remote call. In addition, image objects (see Section 6.6) can contain only transmissible objects. Parameterized types may be transmissible in some instances and not in others; for example, instantiations of the built-in type generators are transmissible only if their parameter types are transmissible. While guardians, creators, and handlers are always transmissible, procedures and iterators are never transmissible.

Users can define new transmissible types. For each transmissible type \( T \) the external representation type of \( T \) must be defined; this describes the format in which objects of type \( T \) are transmitted. Each cluster that implements a transmissible type \( T \) must contain two procedures, encode and decode, to translate objects of type \( T \) to and from their external representation. More information about defining transmissible types can be found in Section 14.

2.5. Orphans

An orphan is an action that has had some ancestor "perish" or has had the pertinent results of some relative action lost in a crash. Orphans can arise in Argus due to crashes and explicit aborts. For example, when a parent action is aborted, the active descendents it leaves behind become orphans. Crashes also cause orphans: when a guardian crashes, all active actions with an ancestor at the crashed guardian and all active actions with committed descendents that ran at the crashed guardian become orphans. However, having a descendent that is an orphan does not necessarily imply that the parent is an orphan; as previously described, actions may commit or abort independently of their subactions.

Argus programmers can largely ignore orphans. Argus guarantees that orphans are aborted before

---

they can view inconsistent data (provided actions are written so that they only communicate through atomic data). Remote cells that fail for any reason may be retired by the system, including cases where the cell action becomes an orphan due to castration (see Section 6.5).

Orphans always abort. They may abort voluntarily or they may be forced to abort by the run-time system; however, an orphan that is in a critical section (executing a action statement, see Section 10.16) may not be forcibly aborted by the run-time system, except by forcing the orphan. On the other hand, the system may encourage orphans (especially orphans that are orphaned) to abort themselves by having their remote cells signal unavailable.

2.6. Deadlocks

Actions in Augus programs may become deadlocked. For example, if action A is waiting for a lock that B holds and B is waiting for a lock that A holds, then A and B are deadlocked. Although implementations may provide some form of deadlock detection or prevention, they are not required to do so. This is because detecting deadlocks is difficult in a language with non-atomic atomic types, since it is not always clear when actions are "waiting" for each other.

If an implementation of Augus chooses to do deadlock detection (presumably for the built-in atomic types), it may only break deadlocks by aborting actions or by restarting operations.
3. Environment

The Argus environment ensures complete static type checking of programs. It also supports separate compilation and the independence of guardians.

3.1. The Library

Argus modules are compiled in the context of a library that gives meaning to external identifiers and allows inter-module type checking. The Argus library contains type information about abstractions; for each abstraction, the library contains a description unit, or DU, describing that abstraction and its implementations. Each DU has a unique name and these names form the basis of type checking.

3.2. Independence of Guardian Images

The code run by a guardian comes from some guardian image. A guardian image contains all the code needed to carry out any local activity of the guardian; any procedure, iterator or cluster used by that guardian will be in its guardian image. Any handler calls made by the guardian, however, are carried out at the called guardian, which contains the code that performs the call. Thus a guardian is independent of the implementations of the guardians it calls and the implementation of a guardian can be changed without affecting the implementations of its clients.

3.3. Guardian Creation

When a guardian is created, it is necessary to select the guardian image that will supply the code run by the new guardian. To this end, each guardian has an associated creation environment that specifies the guardian images for other guardians it may create. The creation environment is a mapping from guardian types to information that can be used to select a guardian image appropriate for each kind of node. For greater flexibility, this information can be associated with particular creator objects.

3.4. The Catalog

Somehow, guardians must be able to find other guardians to call for services. A guardian usually has a reference to any guardian it creates. Also, if a guardian can call some other server guardian, it can learn about the guardians that the server "knows", because guardians can be passed in remote calls. In addition, Argus provides a built-in subsystem known by all guardians. This subsystem is called the catalog. The catalog provides an atomic mapping from names to transmissible objects. For example, when a new guardian is created, it can be catalogued under some well-known name, so that other guardians can find it in the future. Since we are currently experimenting with various interfaces to the catalog, we do not include an interface specification here.
4. Notation

We use an extended BNF grammar to define the syntax of Argus. The general form of a production is:

\[
\text{nonterminal ::= alternative} \\
\hspace{1cm} | \text{alternative} \\
\hspace{1cm} | \text{...} \\
\hspace{1cm} | \text{alternative}
\]

The following extensions are used:

- \( \text{a , ...} \) a list of one or more \( \text{a}'s \) separated by commas: "a" or "a, a" or "a, a, a" etc.
- \( \{ \text{a} \} \) a sequence of zero or more \( \text{a}'s \): " " or "a" or "a a" etc.
- \( [ \text{a} ] \) an optional \( \text{a} \): " " or "a".

Nonterminal symbols appear in normal face. Reserved words appear in bold face. All other terminal symbols are non-alphabetic, and appear in normal face.

Full productions are not always shown in the body of this manual; often alternatives are presented and explained individually. Appendix I contains the complete syntax.
5. Lexical Considerations

A module is written as a sequence of tokens and separators. A *token* is a sequence of "printing" ASCII characters (values 40 octal through 176 octal) representing a reserved word, an identifier, a literal, an operator, or a punctuation symbol. A *separator* is a "blank" character (space, vertical tab, horizontal tab, carriage return, newline, form feed) or a comment. Any number of separators may appear between tokens.

5.1. Reserved Words

The following character sequences are reserved word tokens:

<table>
<thead>
<tr>
<th>abort</th>
<th>else</th>
<th>leave</th>
<th>signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>elseif</td>
<td>mutex</td>
<td>stable</td>
</tr>
<tr>
<td>any</td>
<td>end</td>
<td>nil</td>
<td>string</td>
</tr>
<tr>
<td>array</td>
<td>enter</td>
<td>node</td>
<td>struct</td>
</tr>
<tr>
<td>atomic_array</td>
<td>equates</td>
<td>null</td>
<td>tag</td>
</tr>
<tr>
<td>atomic_record</td>
<td>except</td>
<td>oneof</td>
<td>tagcase</td>
</tr>
<tr>
<td>atomic_variant</td>
<td>exit</td>
<td>others</td>
<td>tagtest</td>
</tr>
<tr>
<td>background</td>
<td>false</td>
<td>own</td>
<td>tagwait</td>
</tr>
<tr>
<td>begin</td>
<td>for</td>
<td>pause</td>
<td>terminate</td>
</tr>
<tr>
<td>bind</td>
<td>foreach</td>
<td>proc</td>
<td>then</td>
</tr>
<tr>
<td>bool</td>
<td>fork</td>
<td>process</td>
<td>topaction</td>
</tr>
<tr>
<td>break</td>
<td>guardian</td>
<td>proctype</td>
<td>transmit</td>
</tr>
<tr>
<td>cand</td>
<td>handler</td>
<td>real</td>
<td>true</td>
</tr>
<tr>
<td>char</td>
<td>handletype</td>
<td>record</td>
<td>type</td>
</tr>
<tr>
<td>cluster</td>
<td>handles</td>
<td>recover</td>
<td>up</td>
</tr>
<tr>
<td>coenter</td>
<td>has</td>
<td>rep</td>
<td>variant</td>
</tr>
<tr>
<td>continue</td>
<td>if</td>
<td>rescignal</td>
<td>when</td>
</tr>
<tr>
<td>cor</td>
<td>image</td>
<td>return</td>
<td>where</td>
</tr>
<tr>
<td>creator</td>
<td>in</td>
<td>returns</td>
<td>while</td>
</tr>
<tr>
<td>creatortype</td>
<td>int</td>
<td>seize</td>
<td>with</td>
</tr>
<tr>
<td>cvt</td>
<td>is</td>
<td>self</td>
<td>wtag</td>
</tr>
<tr>
<td>do</td>
<td>iter</td>
<td>sequence</td>
<td>yield</td>
</tr>
<tr>
<td>down</td>
<td>itertype</td>
<td>signal</td>
<td>yields</td>
</tr>
</tbody>
</table>

Upper and lower case letters are not distinguished in reserved words. For example, 'end', 'END', and 'eNd' are all the same reserved word. Reserved words appear in bold face in this document.

5.2. Identifiers

An *identifier* is a sequence of letters, digits, and underscores (_) that begins with a letter or underscore, and that is not a reserved word. Upper and lower case letters are not distinguished in identifiers.

In the syntax there are two different nonterminals for identifiers. The nonterminal *idn* is used when the identifier has scope (see Section 7.1); *idns* are used for variables, parameters, module names, and as abbreviations for constants. The nonterminal *name* is used when the identifier is not subject to scope rules; names are used for record and structure selectors, oneof and variant tags, operation names, and exceptional condition names.
5.3. Literals

There are literals for naming objects of the built-in types null, bool, int, real, char, and string. Their forms are described in Appendix I.

5.4. Operators and Punctuation Tokens

The following character sequences are used as operators and punctuation tokens.

<table>
<thead>
<tr>
<th>Table 5-2: Operator and Punctuation Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

5.5. Comments and Other Separators

A comment is a sequence of characters that begins with a percent sign (%), ends with a newline character, and contains only printing ASCII characters (including blanks) and horizontal tabs in between. For example:

    z := a[i] + % a comment in an expression
    b[i]

A separator is a blank character (space, vertical tab, horizontal tab, carriage return, newline, form feed) or a comment. Zero or more separators may appear between any two tokens, except that at least one separator is required between any two adjacent non-self-terminating tokens: reserved words, identifiers, integer literals, and real literals. This rule is necessary to avoid lexical ambiguities.
6. Types, Type Generators, and Type Specifications

A type consists of a set of objects together with a set of operations used to manipulate the objects. Types can be classified according to whether their objects are mutable or immutable, and atomic or non-atomic. An immutable object (e.g., an integer) has a value that never varies, while the value (state) of a mutable object can vary over time. Objects of atomic types provide serializability and recovery for accessing actions. Non-atomic types may provide synchronization by specifying that particular operations are executed indivisibly on objects of the type. An operation is indivisible if no other process may affect or observe intermediate states of the operation's execution. Indivisibility properties will be described for all the built-in non-atomic types of Argus.

A type generator is a parameterized type definition, representing a (usually infinite) set of related types. A particular type is obtained from a type generator by writing the generator name along with specific values for the parameters; for every distinct set of legal values, a distinct type is obtained (see Section 12.6). For example, the array type generator has a single parameter that determines the element type; array[int], array[real], and array[array[int]] are three distinct types defined by the array type generator. Types obtained from type generators are called parameterized types or instantiations of the type generator; others are called simple types.

In Argus code, a type is specified by a syntactic construct called a type_spec. The type specification for a simple type is just the identifier (or reserved word) naming the type. For parameterized types, the type specification consists of the identifier (or reserved word) naming the type generator, together with the actual parameter values.

To be used as arguments or results of handler and creator calls, or as image objects (see Section 6.6), objects must be transmissible. Most of the built-in Argus types are transmissible, that is, they have transmissible objects. However, procedures and iterators are never transmissible. For type generators, transmissibility of a particular instantiation of the generator may depend upon transmissibility of any type parameters. A transmissible type provides the pseudo-operation transmit and two internal operations encode and decode. Generally, encode and decode are hidden from clients of the type. They are called implicitly during message transmission (see Section 14) and in creating and decomposing image objects (see Section 6.6). Transmissibility is discussed further in Section 14.

Argus provides all the built-in types of CLU as well as some new types and type generators. This section gives an informal introduction to the built-in types and type generators provided by Argus. Many details are not discussed here, but a complete definition of each type and type generator is given in Appendix II.
6.1. Type Inclusion

The notion of type inclusion in Argus is different from that in CLU. The type any is a type like every other type, and there is no implicit assertion to type any, so there is no need to make a special case for it in the type inclusion rule. Type inclusion in Argus is the same as type equality (see Section 6.2.6), except for procedure, function, handler, and selector types. A routine type O is included in another routine type V, when the number and types of arguments, and the number and types of normal results, are equal, and for each exception in O there is a corresponding exception in V with the same number and types of results. Note that V may have more exceptions than O, and that this rule is not recursive, that is, when comparing types of arguments and results, type identity is used. For example, if we have the following declarations in effect:

\[ p : \text{procedure} \quad \text{and} \quad \text{routine} \quad \text{signature} \{ \text{ovf, undef} \} \]

\[ q : \text{procedure} \quad \text{and} \quad \text{routine} \quad \text{signature} \{ \text{ovf, undef} \} \]

then the type of q is included in the type of p but not vice versa. Thus the assignment \( p \Rightarrow q \) is legal.

6.2. The Sequential Built-in Types and Type Generators

In this section, we introduce the sequential built-in types in Argus. These types are generally the same as types in CLU. This section concentrates on their new characteristics.

Recovery from aborted actions is trivial for immutable objects, since the aborted actions cannot have modified these objects. In particular the built-in vector types such as permanent list, stack, queue, and array are immutable, atomic, and transmittable. The built-in variable type generators included from CLU are not atomic.

6.2.1. Null

The type null has exactly one immutable object, represented by the literal nil, which is atomic and transmittable. See Section 6.1.1 for details.

6.2.2. Bool

The two immutable objects of type bool, with literals true and false, represent logical truth values. The binary operations equal (=), and (\&\&), and or(||), are provided, as well as unary not (!). Objects of type bool are atomic and transmittable. See Section 6.3.3 for details.

6.2.3. Int

The type int models (a range of) the mathematical integers. The exact range is not part of the language definition. Integers are immutable, atomic, transmittable, and their literals are written as a sequence of one or more decimal digits. (There are also several implementation options, see Appendix I.)

However, implementations are encouraged to provide this and other information about the limits of the built-in types in an equate module.

---

\[ \text{signature} \{ \text{ovf, undef} \} \]

\[ \text{procedure} \quad \text{and} \quad \text{routine} \]

\[ p \Rightarrow q \]

\[ \text{permanent list} \]

\[ \text{stack} \]

\[ \text{queue} \]

\[ \text{array} \]

\[ \text{null} \]

\[ \text{null, set, field} \]

\[ \text{null, set, field} \]

\[ \text{null, set, field} \]
The binary operations add (+), sub (−), mul (∗), div (/), mod (mod), power (**), max, and min are provided, as well as unary minus (−) and abs. There are binary comparison operations lt (<), le (<=), equal (=), ge (>=), and gt (>). There are two operations, from_to and from_to_by, for iterating over a range of integers. See Section II.4 for details.

6.2.4. Real

The type real models (a subset of) the mathematical real numbers. The exact subset is not part of the language definition. Reals are immutable, atomic, and transmissible, although transmission of real objects between heterogeneous machine architectures may not be exact. Real literals are written as a mantissa with an optional exponent. A mantissa is either a sequence of one or more decimal digits, or two sequences (one of which may be empty) joined by a period. The mantissa must contain at least one digit. An exponent is 'E' or 'e', optionally followed by '4' or '1', followed by one or more decimal digits. An exponent is required if the mantissa does not contain a period. As is usual, \( m \times 10^e \). Examples of real literals are:

3.14 3.14E0 314e-2 .0314E+2 3. 14

As with integers, the operations add (+), sub (−), mul (∗), div (/), mod (mod), power (**), max, min, minus (−), abs, lt (<), le (<=), equal (=), ge (>=), and gt (>), are provided. It is important to note that there is no form of implicit conversion between types. The l2r operation converts an integer to a real, r2l rounds a real to an integer, and trunc truncates a real to an integer. See Section II.5 for details.

6.2.5. Char

The type char provides the alphabet for text manipulation. Characters are immutable, atomic, transmissible, and form an ordered set. Every implementation must provide at least 128, but no more than 512, characters; the first 128 characters are the ASCII characters in their standard order.

Literals for the printing ASCII characters (octal 40 through octal 176), other than single quote ('), or backslash (\), can be written as that character enclosed in single quotes. Any character can be written by enclosing one of the escape sequences listed in Table 6-1 in single quotes. The escape sequences may be written using upper case letters, but note that escape sequences of the form \( \backslash \) are case sensitive. A table of literals is given at the end of Appendix I. Examples of character literals are:

\'a\', \'\''\', \'\n\', \'\t\', \'\n\', \'\b\', \'\\177\'

There are two operations, l2c and c2l, for converting between integers and characters: the smallest character corresponds to zero, and the characters are numbered sequentially. Binary comparison operations exist for characters based on this numerical ordering: lt (<), le (<=), equal (=), ge (>=), and gt (>). For details, see Section II.6.
### Table 6-1: Character Escape Sequence Forms

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>image</td>
</tr>
<tr>
<td>\t</td>
<td>image</td>
</tr>
<tr>
<td>\r</td>
<td>image</td>
</tr>
<tr>
<td>\v</td>
<td>image</td>
</tr>
<tr>
<td>\a</td>
<td>image</td>
</tr>
<tr>
<td>\f</td>
<td>image</td>
</tr>
<tr>
<td>\m</td>
<td>image</td>
</tr>
<tr>
<td>\x</td>
<td>image</td>
</tr>
<tr>
<td>\c</td>
<td>image</td>
</tr>
<tr>
<td>?</td>
<td>image</td>
</tr>
<tr>
<td>*</td>
<td>image</td>
</tr>
<tr>
<td>#</td>
<td>image</td>
</tr>
</tbody>
</table>

### 6.2.6. String

The type `string` is used for representing text. A string is an immutable, atomic, and transmittable sequence of zero or more characters. Strings are textographically ordered, based on the ordering for characters. A string literal is written as a sequence of zero or more characters or character escape sequences (see Table 6-1), enclosed in double quotes (").

The characters of a string are indexed sequentially starting from one. The `first` operation is used to obtain a character by index. The `subst` operation is used to create a substring. The `all` of a string can be gotten by using `rest`. Searching in strings is provided by the `search` and `match` operations.

Two strings can be concatenated together with `concat()`, and a single character can be appended to the end of a string with `append()`. :cl: can convert a character to a single-character string. The size of a string can be determined with `size`. :cl: scans over the characters of a string, from the first to the last character. There are also the usual homogeneous comparison operators: `if(<), in(=), aequal(=), ge (>=), and gt (>). For details, see Section 11.7.

### 6.2.7. Any

Objects of type `any` may contain objects of any type, and thus provide an escape from compile-time type checking. Unlike CLU, which treats any differently from all other arrays, any is a normal type in Argus. To this end, there is an explicit `create` operation, `create`, and the `free` procedure is also an operation generator of type `any`.

An object of type `any` can be thought of as containing an object and its type. Since there are no operations provided by `type any` that change this state, any objects can be considered to be immutable. However, the state of the contained object may change if that object is created or from this point of view,
the mutability and atomicity of an any object depend on the mutability and atomicity of the contained object. Objects of type any are not transmissible.

The create operation is parameterized by a type; create takes a single argument of that type and returns an any object containing the argument. The force operation is also parameterized by a type; it takes an any and extracts an object of that type, signalling wrong_type if the contained object’s type is not included in the parameter type. The is_type operation is parameterized by a type and checks whether its argument contains an object whose type is included in the parameter type. The detailed specification is found in Section II.19.

6.2.8. Sequence Types

Sequences are immutable and they are atomic or transmissible when instantiated with atomic or transmissible type parameters. Although an individual sequence can have any length, the length and members of a sequence are fixed when the sequence is created. The elements of a sequence are indexed sequentially, starting from one. A sequence type specification has the form:

```
sequence [ type_actual ]
```

where a type_actual is a type_spec, possibly augmented with operation bindings (see Section 12.6).

The new operation returns an empty sequence. A sequence constructor has the form:

```
type_spec $ [ [ expression , ... ] ]
```

and can be used to create a sequence with the given elements.

Although a sequence, once created, cannot be changed, new sequences can be constructed from existing ones by means of the addh, addl, remh, and reml operations. Other operations include fetch, replace, top, bottom, size, the elements and indexes iterators, and subseq. Invocations of the fetch operation can be written using a special form:

```
q[i]
```

% fetch the element at index i of q.

Two sequences with equal elements are equal. The equal(=) operation tests if two sequences have equal elements, using the equal operation of the element type. Similar tests if two sequences have similar elements, using the similar operation of the element type.

All operations are indivisible except for fill_copy, equal, similar, copy, encode, and decode, which are divisible at calls to the operations of the type parameter.

For the detailed specification, see Section II.8.

6.2.9. Array Types

Arrays are one-dimensional, and mutable but not atomic. They are transmissible only if their type parameter is transmissible. The number of elements in an array can vary dynamically. There is no notion of an "uninitialized" element.
The state of an array consists of an integer called the low bound, and a sequence of objects called the elements. The elements of an array are indexed sequentially, starting from the low bound. All of the elements must be of the same type; this type is specified in the array type specification, which has the form:

```
array [ type_actual ]
```

There are a number of ways to create a new array, of which only two are mentioned here. The create operation takes an argument specifying the low bound, and creates a new array with that low bound and no elements. Alternately, an array constructor can be used to create an array with an arbitrary number of initial elements. For example,

```
array[int] 8 [5: 1, 2, 3, 4]
```

creates an integer array with low bound 5, and four elements, while

```
array[bool] 3 [false, true, true]
```

creates a boolean array with low bound 1 (the default), and two elements.

An array type specification states nothing about the bounds of an array. This is because arrays can grow and shrink dynamically, using the add( ), addl( ), and addl( ) operations. Other operations include `fetch( )`, `store( )`, `top( )`, `bottom( )`, `high( )`, `low( )`, the `elements( )` and `invalidation( )` methods. Invocations of `fetch( )` and `store( )` can be written using special forms:

```
a[i]  % fetch the element at index i of a
a[i] := 3  % store 3 at index i of a (by calling store( ))
```

Every newly created array has an identity that is distinct from all other arrays; two arrays can have the same elements without being the same array object. The identity of an array can be distinguished with the `equal( )` operation. The `similar( )` operation tests if two arrays have similar value forms, using the `equal( )` operation of the element type. Similar tests if two arrays have similar values, using the `similar( )` operation of the element type.

All operations are indivisible, except `fill( )`, `similar( )`, `similarl( )`, `copy( )`, `invert( )`, and `decode( )`, which are divisible at calls to operations of the type parameter.

For the detailed specification, see Section II.9.

### 6.2.10. Structure Types

A structure is an immutable collection of one or more named objects. An instantiation is atomic or transmittable only if the type parameters are all exact or all approximations. The names are called selectors, and the objects are called components. Different instantiations may have different types. A structure type specification has the form:

```
struct [ field_spec , ... ]
```

where

```
field_spec ::= name , ... : type_actual
```

Selectors must be unique within a specification, but the ordering and grouping of selectors is unimportant.
6.2.10 Structure Types

A structure is created using a structure constructor. For example, assuming that "info" has been equated to a structure type:

    info = struct[last, first, middle: string, age: int]

the following is a legal structure constructor:

    info $ { last: "Sheaffer", first: "Robert", age: 32, middle: "W." }

An expression must be given for each selector, but the order and grouping of selectors need not resemble the corresponding type specification.

For each selector "sel", there is an operation get_sel to extract the named component, and an operation replace_sel to create a new structure with the named component replaced with some other object. Invocations of the get operations can be written using a special form:

    st.age % get the 'age' component of st

As with sequences, two structures with equal components are in fact the same object. The equal (=) operation tests if two structures have equal components, using the equal operations of the component types. Similar tests if two structures have similar components, using the similar operations of the component types.

All operations are indivisible except for equal, similar, copy, encode, and decode, which are divisible at calls to the operations of the type parameter.

For the detailed specification, see Section II.11.

6.2.11. Record Types

A record is a mutable collection of one or more named objects. Records are never atomic, and are transmissible only if the parameter types are all transmissible. A record type specification has the form:

    record [ field_spec , ... ]

where (as for structures)

    field_spec ::= name , ... : type_actual

Selectors must be unique within a specification, but the ordering and grouping of selectors is unimportant.

A record is created using a record constructor. For example:

    professor $ { last: "Herlhy", first: "Maurice", age: 32, middle: "P." }

For each selector "sel", there is an operation get_sel to extract the named component, and an operation set_sel to replace the named component with some other object. Invocations of these operations can be written using a special form:

    r.middle % get the 'middle' component of r
    r.age := 33 % set the 'age' component of r to 33 (by calling set_age)

As with arrays, every newly created record has an identity that is distinct from all other records; two records can have the same components without being the same record object. The identity of records
can be distinguished with the `equal (=)` operation. The `similar1` operation tests if two records have equal components, using the `equal` operations of the component types. `Similar` tests if two records have similar components, using the `similar` operations of the component types.

All operations are indivisible, except `similar`, `similar1`, `copy`, `encode`, and `decode`, which are divisible at calls to operations of the type parameters.

For the detailed specification, see Section II.12.

### 6.2.12. Oneof Types

A `oneof` type is a tagged, discriminated union. A `oneof` is an immutable labeled object, to be thought of as "one of" a set of alternatives. The label is called the `tag`, and the object is called the `value`. A `oneof` type specification has the form:

```
  oneof [ field_spec , ... ]
```

where (as for structures)

```
  field_spec ::= name , ... : type_actual
```

Tags must be unique within a specification, but the ordering and grouping of tags is unimportant. An instantiation is atomic or transmissible if and only if all the type parameters are atomic or transmissible.

For each tag "t" of a `oneof` type, there is a `make_t` operation which takes an object of the type associated with the tag, and returns the object (as a `oneof`) labeled with tag "t".

To determine the tag and value of a `oneof` object, one normally uses the `tagcase` statement (see Section 10.14).

The `equal (=)` operation tests if two `oneofs` have the same tag, and if so, tests if the two value components are equal, using the `equal` operation of the value type. `Similar` tests if two `oneofs` have the same tag, and if so, tests if the two value components are similar, using the `similar` operation of the value type.

All operations are indivisible, except `equal`, `similar`, `similar1`, `copy`, `encode`, and `decode`, which are divisible at calls to operations of the type parameters.

For the detailed specification, see Section II.14.

### 6.2.13. Variant Types

A `variant` is a mutable `oneof`. Variants are never atomic and are transmissible if and only if their type parameters are all transmissible. A `variant` type specification has the form:

```
  variant [ field_spec , ... ]
```

where (as for `oneofs`)

```
  field_spec ::= name , ... : type_actual
```
6.2.13 Variant Types

The state of a variant is a pair consisting of a label called the tag and an object called the value. For each tag "t" of a variant type, there is a make_t operation which takes an object of the type associated with the tag, and returns the object (as a variant) labeled with tag "t". In addition, there is a change_t operation, which takes an existing variant and an object of the type associated with "t", and changes the state of the variant to be the pair consisting of the tag "t" and the given object. To determine the tag and value of a variant object, one normally uses the tagcase statement (see Section 10.14).

Every newly created variant has an identity that is distinct from all other variants; two variants can have the same state without being the same variant object. The identity of variants can be distinguished using the equal (=) operation. The similar1 operation tests if two variants have the same tag, and if so, tests if the two value components are equal, using the equal operation of the value type. Similar tests if two variants have the same tag, and if so, tests if the two value components are similar, using the similar operation of the value type.

All operations are indivisible, except similar, similar1, copy, encode, and decode, which are divisible at calls to operations of the type parameters.

For the detailed specification, see Section II.15.

6.2.14. Procedure and Iterator Types

Procedures and iterators are created by the Argus system or by the bind expression (see Section 9.8). They are not transmissible. As the identity of a procedure or iterator is immutable, they can be considered to be atomic. However, their atomicity can be violated if a procedure or iterator has own data and thus a mutable state. The immutability and atomicity of a procedure or iterator with own data depends on that operation's specified semantics.

The type specification for a procedure or iterator contains most of the information stated in a procedure or iterator heading; a procedure type specification has the form:

\[
\text{proctype} \left( \left[ \text{type-spec} , \ldots \right] \right) \left[ \text{returns} \right] \left[ \text{signals} \right]
\]

and an iterator type specification has the form:

\[
\text{iterator} \left( \left[ \text{type-spec} , \ldots \right] \right) \left[ \text{yields} \right] \left[ \text{signals} \right]
\]

where

\[
\text{returns} \quad ::= \quad \text{returns} \left( \text{type-spec} , \ldots \right)
\]

\[
\text{yields} \quad ::= \quad \text{yields} \left( \text{type-spec} , \ldots \right)
\]

\[
\text{signals} \quad ::= \quad \text{signals} \left( \text{exception} , \ldots \right)
\]

\[
\text{exception} \quad ::= \quad \text{name} \left[ \left( \text{type-spec} , \ldots \right) \right]
\]

The first list of type specifications describes the number, types, and order of arguments. The returns or yields clause gives the number, types, and order of the objects to be returned or yielded. The signals clause lists the exceptions raised by the procedure or iterator; for each exception name, the number, types, and order of the objects to be returned is also given. All names used in a signals clause must be unique. The ordering of exceptions is not important.
Procedure and iterator types have an `equal(=)` operation. Invocation is not an operation, but a primitive in Argus. For the detailed specification of proctype and iterype, see Section II.17.

6.3. Atomic_Array, Atomic_Record, and Atomic_Variant

Having described the types that Argus inherited from CLU, we now describe the new types in Argus. The mutable atomic type generators of Argus are atomic_array, atomic_record, and atomic_variant. Types obtained from these generators provide the same operations as the analogous types obtained from array, record, and variant, but they differ in their synchronization and recovery properties. Conversion operations are provided between each atomic type generator and its non-atomic partner (for example, atomic_array[i]$aa2a converts from an atomic array to a (non-atomic) array).

An operation of an atomic type generator can be classified as a reader or writer depending on whether it examines or modifies its principal argument, that is, the argument or result object of the operation's type. (For binary operations, such as ar_gets_ar, the operation is classified with respect to each argument.) Intuitively, a reader only examines (reads) the state of its principal argument, while a writer modifies (writes) its principal argument. Operations that create objects of an atomic type are classified as readers. Reader/writer exclusion is achieved by locking: readers acquire a read lock while writers acquire a write lock. The locking rules are discussed in Section 2.2.2.

If one or more of the type parameters is non-atomic, then the resulting type is not atomic because modifications to component objects are not controlled. However, read/write locking still occurs, as described above. Thus, an atomic type generator instantiated with a non-atomic parameter incurs the expense of atomic types without gaining any benefit; such an instantiation is unlikely to be a correct solution to a problem. Atomic type generators yield transmissible types only if the type parameters are all transmissible.

Special operations are provided for each atomic type generator to test and manipulate the locks associated with reader/writer exclusion. These operations are useful for implementing user-defined atomic types (see Section 15). The tagtest and tagwait statements (see Section 10.15) provide additional structured support for atomic variants. The operations can_read, can_write, Test_and_read, and test_and_write provide relatively unstructured access to lock information. For complete definitions of these operations, see Sections II.10, II.13, and II.16.

Assuming normal termination, the following operations acquire read locks on their principal arguments or the objects that they create.

- **atomic_array:**
  - `create`, `new`, `predict`, `fill`, `fill_copy`, `size`, `low`, `high`, `empty`, `top`, `bottom`, `fetch`, `similar`, `similar1`, `copy`, `copy1`, `elements`, `indexes`, `test_and_read`, `a2aa`, `aa2a`, `encode`, `decode`

- **atomic_record:**
  - `create`, `get`, `similar`, `similar1`, `copy`, `copy1`, `test_and_read`, `ar_gets_ar` (second argument), `r2ar`, `ar2r`, `encode`, `decode`

- **atomic_variant:**
  - `make`, `is`, `value`, `av_gets_av` (second argument), `similar`, `similar1`, `copy`, `copy1`, `test_and_read`, `v2av`, `av2v`, `encode`, `decode`
6.3 Atomic_Array, Atomic_Record, and Atomic_Variant

The operations similar and similar1 acquire read locks on both arguments. The operations copy and copy1 acquire a read lock on the value returned as well as their principal argument. Test_and_read is a reader only if it returns true; otherwise it is neither a reader nor a writer.

Assuming normal termination, the following operations acquire write locks on their principal arguments.

- **atomic_array:** set_low, trim, store, addh, addl, remh, reml, test_and_write
- **atomic_record:** set, ar_gets_ar (first argument), test_and_write
- **atomic_variant:** change, av_gets_av (first argument), test_and_write

Test_and_write is a writer only if it returns true; otherwise it is neither a reader nor a writer.

The equal, can_read, and can_write operations are neither readers nor writers.

When an operation of atomic_array terminates with an exception, its principal argument is never modified; however, the atomic_array operations listed above as writers always obtain a write lock before the principal argument is examined, hence there are cases in which they will obtain a write lock and only read, but not modify their principal argument. For example, atomic_array[[trim]] is a writer when it signals bounds. On the other hand, when an atomic_array operation raises a signal because of an invalid argument, no locks are obtained. For example, when atomic_array[[trim]] signals negative_size, it is neither a reader nor a writer since the array’s state is neither examined nor modified (only the integer argument is examined).

For the detailed specification of atomic arrays, see Section II.10; for atomic records, see Section II.13; and for atomic variants, see Section II.16.

6.4. Guardian Types

Guardian types are user-defined types that are implemented by guardian definitions (see Section 13).

A guardian definition has a header of the form:

```
  idn = guardian [ [parms] is idn , ... [ handles idn , ... ] [ where ] ]
```

The creators are the operations named in the identifier list following is; a creator is a special kind of operation that can be called to create new guardians that behave in accordance with the guardian definition. Each guardian optionally provides handlers that can be called to interact with it; the names of these handlers are listed in the identifier list following handles. (See Section 13 for more details.)

A guardian definition named g defines a guardian interface type g. An object of the guardian interface type provides an interface to a guardian that behaves in accordance with the guardian definition. An interface object is created whenever a new guardian is created, and then the interface object can be used to access the guardian’s handlers. Interface objects are transmissible, and after transmission they still give access to the same guardian. In this manual a “guardian interface object” is often called simply a “guardian object”.

The guardian type g for the guardian definition named g has the following operations.
1. The creators listed in the `is` list of the guardian definition.

2. For each handler name `h` listed in the `handles` list, an operation `get_h` with type: `proctype (g) returns (h)`, where `h` is the type of `h`.

3. `Equal` and `similar`, both of type: `proctype (g, g) returns (bool)`, which return true only if both arguments are the same guardian object.

4. `Copy`, of type: `proctype (g) returns (g)`, which simply returns its argument.

5. `transmit`.

A creator may not be named `equal`, `similar`, `copy`, `print`, or `get_h` where `h` is the name of a handler.

Thus if `x` is a variable denoting a guardian interface object of type `g`, and `h` is a handler of `g`, then `g$get_h(x)` will return this handler. As usual with `get_` operations, this call can be abbreviated to `x.h`. Note that the handlers themselves are not operations of the guardian interface type; thus `g$h` would be illegal.

A guardian interface type is somewhat like a structure type. Its objects are constructed by the creators, and decomposed by the `get_` operations. Guardian interface objects are immutable and atomic.

6.5. Handler and Creator Types

Creators are operations of guardian types. Handler objects are created as a side-effect of guardian creation. Unlike procedures and iterators, handlers and creators are transmissible.

The types of handlers and creators resemble the types of procedures:

```
handlertype ([type_spec, ...]) [returns] [signals]
creatertype ([type_spec, ...]) [returns] [signals]
```

The argument, normal result, and exception result types must all be transmissible. The `signals` list for a `handlertype` or `creatertype` cannot include either `failure` or `unavailable`, as these signals are implicit in the interface of all creators and handlers.

Handler and creator types provide `equal` and `similar` operations which return true if and only if both arguments are the same object, and `copy` operations which simply return their argument. For the detailed specification of `handlertype` and `creatertype`, see Section II.18.

6.6. Image

The `image` type provides an escape from compile-time type checking. The main difference between `image` and any is that `image` objects are transmissible. An `image` object can be thought of as a portion of an undecoded message or as the information needed to recreate an object of some type. `image` objects are immutable and atomic.

The `create` operation is parameterized by a transmissible type; it takes a single argument of that type and encodes it (using the `encode` operation of that type) into an `image` object. The `force` operation is also
parameterized by a transmissible type; it takes an Image object and decodes it (using the decode operation of that type) to an object of that type, signalling wrong_type if the encoded object’s type is not included in the parameter type. The is_type operation is parameterized by a type and checks whether its argument is an encoded object of a type included in the parameter type. See Section II.20 for the detailed specification.

6.7. Mutex

Mutex objects are mutable containers for information. They are not atomic, but they provide synchronization and control of writing to stable storage for their contained object. Mutex itself does not provide operations for synchronizing the use of mutex objects. Instead, mutual exclusion is achieved using the seize statement (see Section 10.16), which allows a sequence of statements to be executed while a process is in exclusive possession of the mutex object. Mutex objects are transmissible if the contained object is transmissible.

The type generator mutex has a single parameter that is the type of the contained object. A mutex type specification has the form:

```
mutex [type_actual]
```

Mutex types provide operations to create and decompose mutex objects, and to notify the system of modifications to the mutex object or its contained object.

The create operation takes a single argument of the parameter type and creates a new mutex object containing the argument object. The get_value operation obtains the contained object from its mutex argument, while set_value modifies a mutex object by replacing its contained object. As with records, these operations can be called using special forms, for example:

```
m::mutex[int] := mutex[int]@create (0)
x::int := m.value
m.value := 33 % extract the contained object
% change the contained object
```

Set_value and get_value are indivisible.

Mutexes can be distinguished with the equal (=) operation. There are no operations that could cause or detect sharing of the contained object by two mutexes. Such sharing is dangerous, since two processes would not be synchronized with each other in their use of the contained object if each possessed a different mutex. In general, if an object is contained in a mutex object, it should not be contained in any other object, nor should it be referred to by a variable except when in a seize statement that has possession of the containing mutex.

There are some mutex operations that seize the mutex object automatically. Copy seizes its single argument object. Similar seizes its two argument objects; the first argument object is seized first and then the second. In both cases possession is retained until the operations return. Also, when a mutex object is encoded (for a message or when making an Image), the object is seized automatically. See Section II.21 for the detailed specification of mutex.
Mutexes are used primarily to provide process synchronization and mutual exclusion on shared data, especially to implement user-defined atomic types. In such implementations, it is important to control writing to stable storage. The mutex operation \texttt{changed} provides the necessary control. \texttt{Changed} informs the system that the calling action requires that the argument object be copied to stable storage before the commit of the action's top-level parent (\texttt{topaction}). Any mutex is \texttt{asynchronous}: its contained object is written to stable storage independently of objects that contain that mutex. See Section 15 for further discussion of user-defined atomic objects.

6.8. Node

Objects of type \texttt{node} stand for physical nodes. The operation \texttt{here} takes no arguments and returns the node object that denotes its caller's node. \texttt{Equal}, \texttt{similar}, and \texttt{copy} operations are also provided.

The main use of \texttt{node} objects is in guardian creation (see Section 13), where they are used to cause a newly created guardian to reside at a particular node. Objects of type \texttt{node} are immutable, atomic, and transmissible. For the detailed specification, see Section 11.2.

6.9. Other Type Specifications

A type specification for a user-defined type has the form of a \texttt{reference}:

\begin{equation}
\text{reference ::= idn} \\
\quad \mid \text{idn [ actual_parm , ... ]} \\
\quad \mid \text{reference $ name}
\end{equation}

where each \texttt{actual_parm} must be a compile-time computable constant (see Section 7.2) or a \texttt{type_actual} (see Section 12.6). A reference must denote a data abstraction to be used as a type specification; this syntax is provided for referring to a data abstraction that is named in an \texttt{equate} module (see Section 12.4). For type generators, actual parameters of the appropriate types and number must be supplied. The order of parameters is always significant for user-defined types (see Section 12.5).

There are two special type specifications that are used when implementing new abstractions: \texttt{rep}, and \texttt{cvt}. These forms may only be used within a cluster; they are discussed further in Section 12.3.

Within an implementation of an abstraction, formal parameters declared with \texttt{type} can be used as type specifications. Finally, identifiers that have been equated to type specifications can also be used as type specifications.
7. Scopes, Declarations, and Equates

This section describes how to introduce and use constants and variables, and the scope of constant and variable names. Scoping units are described first, followed by a discussion of variables, and finally constants.

7.1. Scoping Units

Scoping units follow the nesting structure of statements. Generally, a scoping unit is a body and an associated "heading". The scoping units are as follows (see Appendix I for details of the syntax).

1. From the start of a module to its end.
2. From a cluster, proc, iter, equates, guardian, handler, or creator to the matching end.
3. From a for, do, begin, background, recover, enter, coenter, or seize to the matching end.
4. From a then or else in an if statement to the end of the corresponding body.
5. From a tag, wtag, or others in a tagcase, tagwait, or tagtest statement to the end of the corresponding body.
6. From a when or others in an except statement to the end of the corresponding body.
7. From the start of a type_set to its end.
8. From an action or topaction to the end of the corresponding body.

The structure of scoping units is such that if one scoping unit overlaps another scoping unit (textually), then one is fully contained in the other. The contained scope is called a nested scope, and the containing scope is called a surrounding scope.

New constant and variable names may be introduced in a scoping unit. Names for constants are introduced by equates, which are syntactically restricted to appear grouped together at or near the beginning of scoping units (except in type sets). For example, equates may appear at the beginning of a body, but not after any statements in the body.

In contrast, declarations, which introduce new variables, are allowed wherever statements are allowed, and hence may appear throughout a scoping unit. Equates and declarations are discussed in more detail in the following two sections.

In the syntax there are two distinct nonterminals for identifiers: idn and name. Any identifier introduced by an equate or declaration is an idn, as is the name of the module being defined, and any operations it has. An idn names a specific type or object. The other kind of identifier is a name. A name is generally used to refer to a piece of something, and is always used in context; for example, names are used as record selectors. The scope rules apply only to idns.

The scope rules are simple:
1. An idn may not be redefined in its scope.
2. Any idn that is used as an external reference in a module may not be used for any other purpose in that module.
Unlike other "block-structured" languages, Argus prohibits the redefinition of an identifier in a nested scope. An identifier used as an external reference names a module or constant; the reference is resolved using the compilation environment.

7.1.1. Variables

Variables are the fundamental "things" in the Argus universe; variables are a mechanism for denoting (i.e., naming) objects. A variable has three properties: its type, whether it is stable or not, and the object that it currently denotes (if any). A variable is said to be uninitialized if it does not denote any object. Attempts to use uninitialized variables are programming errors and (if not detected at compile-time) cause the guardian to crash.

There are only three things that can be done with variables:

1. New variables can be introduced. Declarations perform this function, and are described below.

2. An object may be assigned to a variable. After an assignment the variable denotes the object assigned.

3. A variable may be used as an expression. The value of a variable is the object that the variable denotes at the time the expression is evaluated.

7.1.2. Declarations

Declarations introduce new variables. The scope of a variable is from its declaration to the end of the smallest scoping unit containing its declaration; hence, variables must be declared before they are used.

There are two sorts of declarations: those with initialization, and those without. Simple declarations (those without initialization) take the form

\[ \text{decl} ::= \text{idn}, ..., : \text{type spec} \]

A simple declaration introduces a list of variables, all having the type given by the \text{type spec}. This type determines the types of objects that can be assigned to the variable. The variables introduced in a simple declaration initially denote no objects, i.e., they are uninitialized.

A declaration with initialization combines declarations and assignments into a single statement. A declaration with initialization is entirely equivalent to one or more simple declarations followed by an assignment statement. The two forms of declaration with initialization are:

\[ \text{idn} : \text{type spec} ::= \text{expression} \]

and

\[ \text{decl}_1, ..., \text{decl}_n ::= \text{call} \]

These are equivalent to (respectively):

\[ \text{idn} : \text{type spec} \]

\[ \text{idn} ::= \text{expression} \]

and
7.1.2 Declarations

\[\text{decl}_{1} \ldots \text{decl}_{n} \quad \% \text{ declaring idn}_{1} \ldots \text{idn}_{m}\]
\[\text{idn}_{1}, \ldots, \text{idn}_{m} := \text{call}\left[@\text{primary}\right]\]

In the second form, the order of the ids in the assignment statement is the same as in the original declaration with initialization. (The call must return \(m\) objects.)

7.2. Equates and Constants

An equate allows an identifier to be used as an abbreviation for a constant, type set, or equate module name that may have a lengthy textual representation. An equate also permits a mnemonic identifier to be used in place of a frequently used constant, such as a numerical value. We use the term constant in a very narrow sense here: constants, in addition to being immutable, must be computable at compile-time. Constants are either types (built-in or user-defined), or objects that are the results of evaluating constant expressions. (Constant expressions are defined below.)

The syntax of equates is:

\[
\text{equate} ::= \text{idn} = \text{constant} \\
| \text{idn} = \text{type set} \\
| \text{idn} = \text{reference}
\]

\[
\text{constant} ::= \text{type spec} \\
| \text{expression}
\]

\[
\text{type set} ::= \{ \text{idn} | \text{idn}\ \text{has}\ \text{oper}\ \text{decl}, \ldots \} \{\text{equate}\}
\]

\[
\text{reference} ::= \text{idn} \\
| \text{idn} [\text{actual parm}, \ldots ] \\
| \text{reference }$\text{name}$
\]

References can be used to name equate modules.

An equated identifier may not be used on the left-hand side of an assignment statement.

The scope of an equated identifier is the smallest scoping unit surrounding the equate defining it; here we mean the entire scoping unit, not just the portion after the equate. All the equates in a scoping unit must appear grouped near the beginning of the scoping unit. The exact placement of equates depends on the containing syntactic construct; usually equates appear at the beginnings of bodies.

Equates may be in any order within the a scoping unit. Forward references among equates in the same scoping unit are allowed, but cyclic dependencies are illegal. For example,

\[
x = y \\
y = z \\
z = 3
\]

is a legal sequence of equates, but
\[ x = y \\
y = z \\
z = x \]

is not. Since equates introduce ids, the scoping restrictions on ids apply (i.e., the id may not be defined more than once).

### 7.2.1. Abbreviations for Types

Identifiers may be equated to type specifications, giving abbreviations for type names.

### 7.2.2. Constant Expressions

We define the subset of objects that equated identifiers may denote by stating which expressions are constant expressions. (Expressions are discussed in detail in Section 8.) A constant expression is an expression that can be evaluated at compile-time to produce an immutable object of a built-in type. This includes:

1. Literals.
2. Identifiers equated to constants.
3. Formal parameters.
4. Procedure, iterator, and creator names.
5. Bind expressions (see Section 9.0), where the routine bound and the explicit arguments are all constants.
6. Invocations of procedure operations on the built-in immutable types, provided that all operands are constant expressions that are not formal parameters.

The built-in immutable types are: null, int, real, bool, char, string, sequence types, exact types, structure types, procedure types, iterator types, and creator types.

We explicitly forbid the use of formal parameters as operands to calls in constant expressions, since the values of formal parameters are not known at compile-time. If the evaluation of a constant expression would signal an exception, the constant defined by that expression is illegal.
8. Assignment and Calls

The two fundamental activities of Argus programs are calls and assignment of computed objects to variables.

Argus programs should use mutual exclusion or atomic data to synchronize access to all shared variables, because Argus supports concurrency and thus processes can interfere with each other during assignments. For example,

\[
\begin{align*}
  i &= 1 \\
  j &= 2
\end{align*}
\]

is not equivalent to

\[
\begin{align*}
  i, j &= 1, 2
\end{align*}
\]

in the presence of concurrent assignments to the same variables, because any interleaving of indivisible events is possible in the presence of concurrency.

Argus is designed to allow complete compile-time type-checking. The type of each variable is known by the compiler. Furthermore, the type of objects that could result from the evaluation of any expression is known at compile time. Hence, every assignment can be checked at compile time to ensure that the variable is only assigned objects of its declared type. An assignment \( v := E \) is legal only if the type of \( E \) is included the type of \( v \). The definition of type inclusion is given in Section 6.1.

8.1. Assignment

Assignment causes a variable to denote an object. Some assignments are implicitly performed as part of the execution of various mechanisms of the language (in exception handling, and the tagcase, tagtest, and tagwait statements). All assignments, whether implicit or explicit, are subject to the type inclusion rule.

8.1.1. Simple Assignment

The simplest form of assignment statement is:

\[
\text{idn} := \text{expression}
\]

In this case the expression is evaluated, and then the resulting object is assigned to the variable named by the idn in an indivisible event. Thus no other process may observe a "half-assigned" state of the variable, but another process may observe various states during the expression evaluation and between the evaluation of the expression and the assignment. The expression must return a single object (whose type must be included in that of the variable).

8.1.2. Multiple Assignment

There are two forms of assignment statement that assign to more than one variable at once:

\[
\text{idn}, \ldots := \text{expression}, \ldots
\]

and
idn, ..., := call [ @ primary ]

The first form of multiple assignment is a generalization of simple assignment. The first variable is assigned the first expression, the second variable the second expression, and so on. The expressions are all evaluated (from left to right) before any assignments are performed. The assignment of multiple objects to multiple variables is an indivisible event, but evaluation of the expressions is divisible from the actual assignment. The number of variables in the list must equal the number of expressions, no variable may occur more than once, and the type of each variable must include the type of the corresponding expression.

The second form of multiple assignment allows one to retain the objects resulting from a call returning two or more objects. The first variable is assigned the first object, the second variable the second object, and so on, but all the assignments are carried out indivisibly. The order of the objects is the same as in the return statement executed in the called routine. The number of variables must equal the number of objects returned, no variable may occur more than once, and the type of each variable must include the corresponding return type of the called procedure.

8.2. Local Calls

In this section we discuss procedure calls; iterator calls are discussed in Section 10.12. However, argument passing is the same for both procedures and iterators.

Local calls take the form:

primary ([ expression, ... ])

The sequence of activities in performing a local call are as follows:
1. The primary is evaluated.
2. The expressions are evaluated, from left to right.
3. New variables are introduced corresponding to the formal arguments of the routine being called (i.e., a new environment is created for the called routine to execute in).
4. The objects resulting from evaluating the expressions (the actual arguments) are assigned to the corresponding new variables (the formal arguments). The first formal is assigned the first actual, the second formal the second actual, and so on. The type of each expression must be included in the type of the corresponding formal argument.
5. Control is transferred to the routine at the start of its body.

A call is considered legal in exactly those situations where all the (implicit) assignments are legal.

A routine may assign an object to a formal argument variable; the effect is just as if that object were assigned to any other variable. From the point of view of the called routine, the only difference between its formal argument variables and its other local variables is that the formals are initialized by its caller.

Procedures can terminate in two ways: they can terminate normally, returning zero or more objects, or they can terminate exceptionally, signalling an exceptional condition. When a procedure terminates
normally, any result objects become available to the caller, and can be assigned to variables or passed as
arguments to other routines. When a procedure terminates exceptionally, the flow of control will not go to
the point of return of the call, but rather will go to an exception handler (see Section 11).

8.3. Handler Calls

As explained in Section 2 and in Section 13, a handler is an operation that belongs to some guardian.
A handler call causes an activation of the called handler to run at the handler’s guardian; the activation is
performed at the called handler’s guardian by a new subaction created solely for this purpose. Usually
the handler’s guardian is not the same as the one in which the call occurs, and the called handler’s
guardian is likely to reside at a different node in the network than the calling guardian. However, it is legal
to call a handler that belongs to a guardian residing at the caller’s node, or even to call a handler
belonging to the caller’s guardian.

Although the form of a handler call looks like a procedure call:

```
primary ([ expression, ... ])
```

its meaning is very different. Among other things, a handler is called remotely, with the arguments and
results being transmitted by value in messages, and the call is run as a subaction of its calling action.
Below we present an overview of what happens when executing a handler call and then a detailed
description.

A handler call runs as a subaction of the calling action. We will refer to this subaction as the call action.
The first thing done by the call action is the transmission of the arguments of the call. Transmission is
accomplished by encoding each argument object, using the encode operation of its type. The arguments
are decoded at the called guardian by a subaction of the call action called the activation action. Each
argument is decoded by using the decode operation of its type. The effect of transmission is that the
arguments are passed by value from the caller to the handler activation: new objects come into existence
at the handler’s guardian that are copies of the argument objects. Object values are transmitted in such a
way as to preserve the internal sharing structure of each argument object is preserved\(^6\), as well as any
sharing structure between the argument objects in a single call. See Section 14 for further discussion of
transmission.

After the arguments have been transmitted, the activation action performs the handler body. When the
handler body terminates, by executing a return, abort return, signal, or abort signal statement, the
result objects are transmitted to the caller by encoding them at the handler’s guardian, and committing or
aborting the activation action (as it specified). The call action then decodes the results at the caller’s
guardian. Once the results have been transmitted to the caller, the call action commits and execution
continues in the caller as indicated by the caller’s code. (Note that the call action will commit even if the
activation action aborts.)

---

\(^6\)This is only strictly true for the built-in types. A user-defined type might not preserve internal sharing structure.
The above discussion has ignored the possibility of several problems that may arise in executing a handler call. These problems either cause the call action or the activation action to abort or result in the crash of the calling guardian. A handler call attempt may cause a separation or expansion in a programming error, and so if this happens the calling guardian is damaged. Other problems cause the call action or the activation action to be aborted, and this is communicated to the caller as an exception raised by the Argus system. Two such exceptions can be raised: failure (string) and unavailable (string). The string exception result summarizes the problem that has occurred.

The meaning of a failure exception generated by the Argus system is that this particular call did not succeed, and furthermore it is unlikely to succeed if reattempted. There are two reasons why failure is raised: an error occurred in transmitting an argument or result, or the handler's guardian no longer exists.

The Argus system raises the unavailable exception when it is unable to communicate with the handler's guardian. Reasons why communication may fail include network problems and a crash of the called guardian or its node. The Argus system raises the unavailable exception only if communication seems impossible at that time; it may try many times to establish communication. Therefore, when a call terminates with the unavailable exception, there is little chance to successfully complete the call immediately. However, unlike a call terminated by the failure exception, a call terminated by the unavailable exception may complete successfully if raised later. Note that the arguments and results may be executed several times as the system tries to establish communication.

For example, suppose we have a handler call:

```
m.send_mail(user, my_message)
```

where m is a master guardian, and the send_mail handler has the handler

```
send_mail = handler (n: user_id, msg: message, client: user, user_id)
```

Then user and my_message are assumed to be user_id and message, respectively, and the created values are assumed to be created in the default handler by the default operations of these types. If user is actually registered to receive mail, then the call proceeds normally; otherwise it signals no_such_user. In either case no executing or deciding of the call is needed since there is no result.

Possible exceptions from this call are no_such_user, failure, and unavailable. So the call might be performed in an except statement:

```
m.send_mail(user, my_message)
except when no_such_user: ...
    when unavailable (n: string): ...
    when failure (n: string): ...
end
```
8.3.1 Semantics of Handler Calls

In this section we describe the semantics of a handler call in detail. A handler call causes activity at both the calling guardian and at the called guardian. At the calling guardian, the sequence of activities in performing a handler call is as follows:

1. The primary is evaluated.
2. The argument expressions are evaluated from left to right.
3. A subaction, which we will refer to as the call action, is created for the remote call. All subsequent activity on behalf of the call will be performed by the call action or one of its descendants. For it to be possible to create the call action, the caller must already be running as an action. Remote calls by non-actions are programming errors and cause the calling guardian to crash.
4. A call message is constructed. As part of constructing this message, encode operations are performed on the argument objects. If any of the encode operations terminates with a failure exception, then the remote call will terminate with the same exception, and the call action will be aborted.
5. The call message is sent to the guardian of the called handler, and the call action waits for the completion of the call.
6. If the call message arrives at the node of the target guardian, and the target guardian does not exist, then the call action is aborted with the failure exception having the string "guardian does not exist" as its exception result.
7. If the system determines that it cannot communicate with the called guardian, it aborts the call action. The call action may be retried several times (beginning at step 3) in attempts to communicate. If repeated communication failures are encountered, the system aborts the call action and causes the call to terminate with the unavailable exception. The system will cause this kind of termination only when it is extremely unlikely that retrying the call immediately will succeed.
8. Ordinarily, a call completes when a reply message containing the results is received. When the reply message arrives at the caller, it is decoded using the decode operation for each result object. If any decode terminates with a failure exception, the call action is aborted, and the call terminates with the same exception. Otherwise, the call action commits.
9. The call will terminate normally if the result message indicates that the handler activation returned (instead of signalled); otherwise it terminates with whatever exception was signalled.

At the called guardian, the following activities take place.

1. A subaction of the call action is created at the target guardian to run the call. We will refer to this subaction as the activation action. All activity at the target guardian occurs on behalf of the activation action or one of its descendants.
2. The call message is decomposed into its constituent objects. As part of this process decode operations are performed on each argument. If any decode terminates with a failure exception, then the activation action is aborted, and the call terminates with the same exception.
3. The called handler is called within the activation action. This call is like a regular procedure call. The objects obtained from decoding the message are the actual arguments, and they are bound to the formals via implicit assignments.
4. If the handler terminates by executing an abort return or an abort signal statement (see Section 11.1), then all committed descendants of the activation action are aborted. Then the reply message is constructed by encoding the result objects, the activation action is
aborted, and the reply message is sent to the caller. Otherwise, when the handler terminates, the reply message is constructed by encoding the result objects, the activation action commits, and the reply message is sent to the caller. If one of the calls of encode terminates with a failure exception, then the activation action is aborted, and the call terminates with the same exception.

When the Argus system terminates a call with the unavailable exception, it is possible that the activation action and/or some of its descendants are actually running. This could happen, for example, if the network partitions. These running processes are called "orphans". The Argus system makes sure that orphans will be aborted before they can view inconsistent data (see Section 2.5).

8.4. Creator Calls

Creators are called to cause new guardians to come into existence. As part of the call, the node at which the newly created guardian will be located may be specified. If the node is not specified, then the new guardian is created at the same node as the caller of the creator. The form of a creator call is:

```
primary ( [ expression, ... ] ) [ @ primary ]
```

The *primary* following the at-sign (@) must be of type node.

A creator call causes two activities to take place. First, a new guardian is created at the indicated node. Second, the creator is called as a handler at the newly created guardian. This handler call has basically the same semantics as the regular handler call described above.

The Argus system may also cause a creator call to abort with the failure or unavailable exceptions. The reasons for such terminations are the same as those for handler calls, and the meanings are the same: the failure exception means that the call should not be retried, while the unavailable exception means that the call should not be retried immediately.

8.4.1. Semantics of Creator Calls

The activities carried out in executing a creator call are as follows.

1. The (first) *primary* is evaluated.

2. The argument *expressions* are evaluated from left to right.

3. The optional *primary* following the at-sign is evaluated to obtain a node object. If this *primary* is missing, the node at which the call is taking place is used.

4. A subaction, which we will refer to as the *call* action, is created. All subsequent activity takes place within this subaction. As was the case for handler calls, creators can be called only from within actions. A creator call by a non-action is a programming error and causes the calling guardian to crash.

5. A new guardian is created at the indicated node. The creator obtained in step 1 will indicate the type of this guardian. The selection of a particular load image for this type will occur as discussed in Section 3.3.

6. As was the case for handler calls, if the system cannot communicate with the indicated node, the creator call will terminate with the unavailable exception. If the system is unable
8.4.1 Semantics of Creator Calls

to determine what implementation to load, or if there is no implementation of the type that
can run on the indicated node, or if the manager of the node refuses to allow the new
guardian to be created, the creator call will terminate with the failure exception. In either
case the call action will be aborted.

7. A remote call is now performed to the creator. This call has the same semantics as
described for handler calls above in steps 4 through 9 of the activities at the calling node
and also steps 1 through 4 of activities at the called node. However, if either the call action
or the activation action aborts, the newly created guardian will be destroyed.

For example, suppose we execute the creator call

\[ x: G := \text{G$create(3 \, @ n}$ \]

where \( G \) is a guardian type, \( n \) denotes an object of type node, and \( \text{create} \) has header

\[ \text{create} = \text{creator(\( n: \text{Int} \)} \text{ returns (} G \text{) signals (} \text{not\_possible(} \text{string} \text{))} \]

The system will select an implementation of \( G \) that is suitable for use at node \( n \), and will then create a
guardian at node \( n \) running that implementation. Next \( \text{create}(3) \) is performed as a handler call at that
new guardian. If \( \text{create} \) returns, then the assignment to \( x \) will occur, causing \( x \) to refer to the new
guardian that \( \text{create} \) returned; now we can call the handlers provided by \( G \). The exceptions that can be
signalled by this call are \( \text{not\_possible}, \text{failure}, \text{and unavailable} \). An example of a call that handles all
these exceptions is:

\[ x: G := \text{G$create(3 \, @ n}$ \]

\[ \text{except when not\_possible(s: string)}: \ldots \]
\[ \text{when failure(s: string)}: \ldots \]
\[ \text{when unavailable(s: string)}: \ldots \]
\[ \text{end} \]

Creators are described in more detail in Section 13.
9. Expressions

An expression evaluates to an object in the Argus universe. This object is said to be the result or value of the expression. Expressions are used to name the object to which they evaluate. The simplest forms of expressions are literals, variables, parameters, equated identifiers, equate module references, procedure, iterator, and creator names, and self. These forms directly name their result object. More complex expressions are built up out of nested procedure calls. The result of such an expression is the value returned by the outermost call.

9.1. Literals

Integer, real, character, string, boolean and null literals are expressions. The type of a literal expression is the type of the object named by the literal. For example, true is of type bool, "abc" is of type string, etc. (see the end of Appendix I for details).

9.2. Variables

Variables are identifiers that denote objects of a given type. The type of a variable is the type given in the declaration of that variable. An attempt to use an uninitialized variable as an expression is a programming error and causes the guardian to crash.

9.3. Parameters

Parameters are identifiers that denote constants supplied when a parameterized module is instantiated (see Section 12.5). The type of a parameter is the type given in the declaration of that parameter. Type parameters cannot be used as expressions.

9.4. Equated Identifiers

Equated identifiers denote constants. The type of an equated identifier is the type of the constant which it denotes. Identifiers equated to types, type_sets, and equate modules cannot be used as expressions.

9.5. Equate Module References

Equate modules provide a named set of equates (see Section 12.4). To use a name defined in an equate module as an expression, one writes:

```
reference $ name
```

where

```
reference ::= idn

\| idn [ actual_parm , ... ]
\| reference $ name
```

The type of a reference is the type of the constant which it denotes. Identifiers equated to types, type_sets, and equate modules cannot be used as expressions.
9.6. Self

The expression self evaluates to the object (or guardian block) corresponding to the guardian instance within which the expression is evaluated. A self expression may appear syntactically within the body of a guard. See Section 10 for further discussion.

9.7. Procedure, Function, and Creator Names

A procedure and function name is a name that denotes a routine nested within a cluster. Creators may only be defined within a guardian module. These names in a guardian module are named by expressions of the form:

```
    idn ((actual_param, ...))
```

The actual parameters of a parameterized procedure or function can be either constants or type actuals (see Section 12.6).

When a procedure, function, or creator is defined as an expression of a type, that type is part of the name of the routine. The form for naming an expression of type:

```
    type_spec & name ((actual_param, ...))
```

The type of this expression is the type of the named routine.

9.8. Bind

Clauses may be created by the bind expression:

```
    bind entity ((bind_args, ...))
```

where

```
bind_args ::= *
  | expression

entity ::= reference
  | entity * entity
  | entity & entity
  | entity (entity_arg, ...)
  | entity [entity_arg, ...]
  | entity (entity_arg, ...))
  | (entity_arg, ...)
  | (entity_arg, ...
  | (entity_arg, ...
  | (entity_arg, ...

An entity is a simple kind of expression that is used to prevent implicit ambiguity.

The number of bind_args must match the entity's parameter count. A bind_args that is an asterisk (*) indicates an expression containing an entity. A bind_args that is a fieldname indicates that the type of the expression must have been declared to be of type entity. When a fieldname is used as a whole in a method-type expression, it indicates that the expression is named as a selector in that the entity's bound.
The evaluation of a bind expression proceeds by first evaluating the entity and then evaluating, from left to right, any bind_args that are expressions. The entity may evaluate to a procedure, iterator, handler, or creator object. Suppose that the entity is a procedure or iterator object. (Creator and handler bindings are discussed below.) Then the result is formed by binding the argument objects to the corresponding formals of the entity to form a closure; note that the procedure or iterator is not called when the bind expression is evaluated. When the closure is called, the object denoted by the entity is passed all the bound objects and any actual arguments supplied in the call, all in the corresponding argument positions.

For example, suppose we have:

\[
p = \text{proc}(x: T, y: \text{int}, w: S) \rightarrow \text{returns}(R) \rightarrow \text{signals}(\text{too_big})
\]

Then

\[
q := \text{bind} \; p(\ast, 3 + 4, \ast)
\]

produces a procedure whose type is \(\text{proctype}(T, S) \rightarrow \text{returns}(R) \rightarrow \text{signals}(\text{too_big})\) and assigns it to \(q\). A call of \(q(a, b)\) is then equivalent to the call \(p(a, 7, b)\).

Bound routines will be stored in stable storage if they are accessible from a stable variable (see Section 13.1). In this case the entity and the bind_args should denote atomic objects.

There is only one instance of a routine's own data for each parameterization; thus all the bindings of a routine share its own data, if any (see Section 12.7). Each binding is generally a new object; thus the relevant equal operation may treat syntactically identical bindings as distinct.

The semantics of binding a creator or handler are similar to binding a procedure or iterator; the differences arise from argument transmission. Encoding of bound argument objects happens when the bind expression is evaluated and sharing is only preserved among objects bound at the same time (see Section 14). In more detail, the evaluation of a bind expression proceeds by first evaluating the entity and then evaluating, from left to right, any bind_args that are expressions. Then the argument objects are encoded, from left to right, preserving sharing among these objects. The result is formed by binding the encoded argument objects to the corresponding formals of the entity to form a closure. Note that the entity is not called when the bind expression is evaluated.

When the closure is called, first any other arguments are evaluated and encoded (not sharing with the bound objects) and then the call to the entity is initiated. Decoding of the arguments at the called guardian is done in reverse of the order of encoding; that is, other arguments are decoded before bound arguments and the most recently bound arguments are decoded first. Sharing is preserved on decoding only among groups of bound arguments and among the other arguments, not between groups. Thereafter the call proceeds as normally.

For example, if we execute

\[
h1 := \text{bind} \; h(x, y, \ast)
\]

\(h1(z)\)
then sharing of objects between $x$ and $y$ will be preserved by transmission, but sharing will not be preserved between $x$ and $z$ or $y$ and $z$.

Closures can be used in equates, provided all the expressions are constants (see Section 7.2.2). However, a handler cannot appear in an equate, since it is not a constant.

9.9. Procedure Calls

Procedure calls have the form:

```
primary ([ expression, ... ])
```

The primary is evaluated to obtain a procedure object, and then the expressions are evaluated left to right to obtain the argument objects. The procedure is called with these arguments, and the object returned is the result of the entire expression. For more discussion see Section 8.

Any procedure call $p(E_1, ..., E_n)$ must satisfy two constraints to be used as an expression: the type of $p$ must be of the form:

```
proctype (T_1, ..., T_n) returns (R) signals (...)
```

and the type of each expression $E_i$ must be included in the corresponding type $T_i$. The type of the entire call expression is given by $R$.

9.10. Handler Calls

Handler calls have the form:

```
primary ([ expression, ... ])
```

The primary is evaluated to obtain a handler object, and then the expressions are evaluated left to right to obtain the argument objects. The handler is then called with these arguments as discussed in Section 8.3. The following expressions are examples of handler calls:

```
h(x)
info_guard.who_is_user("john", "doe")
dow_jones.into("XYZ Corporation")
```

Any handler call $h(E_1, ..., E_n)$ must satisfy the following constraints when used as an expression. The type of $h$ must be of the form:

```
handlertype (T_1, ..., T_n) returns (R) signals (...)
```

and the type of each expression $E_i$ must be included in the corresponding type $T_i$. The type of the entire call expression is given by $R$.

As explained in Section 8.3, the execution of a handler call starts by creating a subaction. Therefore an attempt to call a handler from a process that is not running an action is a programming error and will cause the calling guardian to crash. This crash occurs after all of the component expressions have been evaluated.
9.11. Creator Calls

Creator calls have the form:

```
primary ([ expression, ... ]) [@ primary]
```

The first `primary` is evaluated to obtain a creator object, the argument expressions are evaluated left to right to obtain the argument objects, and then the `primary` following the at-sign (@), if present, is evaluated to obtain a node object. If the `primary` following the at-sign is omitted, then `node$here()` is used. The guardian is then created at that node, and the creator called, as discussed in Section 8.4. The following are examples of creator calls:

```
mailer$create() @ n
spooler(devtype)$create()
```

A creator call `c(E_1,...,E_n)@n` must satisfy the following constraints when used as an expression. The type of `c` must be of the form:

```
creattypename (T_1,...,T_n) returns (R) signals (...) 
```

where each `T_i` includes the type of the corresponding expression `E_i`. `n` must be of type `node`. The type of the entire call expression is given by `R`.

As with handler calls, an attempt to call a creator from a process that is not running an action will cause the calling guardian to crash after all component expressions have been evaluated.

9.12. Selection Operations

Selection operations provide access to the individual elements or components of a collection. Simple notations are provided for calling the `fetch` operations of array-like types, and the `get` operations of record-like types. In addition, these "syntactic sugarings" for selection operations may be used for user-defined types with the appropriate properties.

9.12.1. Element Selection

An element selection expression has the form:

```
primary [ expression ]
```

This form is just syntactic sugar for a call of a fetch operation, and is computationally equivalent to:

```
T$fetch(primary, expression)
```

where `T` is the type of the `primary`. `T` must provide a procedure operation named `fetch`, which takes two arguments whose types include the types of `primary` and `expression`, and which returns a single result.

9.12.2. Component Selection

The component selection expression has the form:

```
primary . name
```

This form is just syntactic sugar for a call of a `get_name` operation, and is computationally equivalent to:

```
T$get_name(primary)
```

where `T` is the type of `primary`. `T` must provide a procedure operation named `get_name`, that takes one
argument and returns a single result. Of course, the type of the procedure's argument must include the type of the primary.

9.13. Constructors
Constructors are expressions that enable users to create and initialize sequences, arrays, atomic arrays, structures, records, and atomic records. There are no constructors for user-defined types.

9.13.1. Sequence Constructors
A sequence constructor has the form:

\[ \text{type_spec} \{ \left[ \text{expression} , \ldots \right] \} \]

The \text{type_spec} must name a sequence type: \text{sequence}[T]. This is the type of the constructed sequence. The expressions are evaluated to obtain the elements of the sequence. They correspond (left to right) to the indexes 1, 2, 3, etc. For a sequence of type \text{sequence}[T], the type of each element expression in the constructor must be included in \text{T}.

A sequence constructor is computationally equivalent to a sequence \text{new} operation, followed by a number of sequence \text{addh} operations.

9.13.2. Array and Atomic Array Constructors
An array or atomic array constructor has the form:

\[ \text{type_spec} \{ \left[ \text{expression} : \right] \left[ \text{expression} , \ldots \right] \} \]

The \text{type_spec} must name an array or atomic array type: \text{array}[T] or \text{atomic_array}[T]. This is the type of the constructed array. The optional expression preceding the colon (:) must evaluate to an integer, and becomes the low bound of the constructed array or atomic array. If this expression is omitted, the low bound is 1. The optional list of expressions is evaluated to obtain the elements of the array. These expressions correspond (left to right) to the indexes \text{low_bound}, \text{low_bound}+1, \text{low_bound}+2, etc. For an array or atomic array of type \text{array}[T] or \text{atomic_array}[T], the type of each element expression in the constructor must be included in \text{T}. A constructor of the form \text{array}[T][0] has a low bound of 1 and no elements.

An array constructor is computationally equivalent to a \text{create} operation, followed by a number of \text{addh} operations.

9.13.3. Structure, Record, and Atomic Record Constructors
A structure, record, or atomic record constructor has the form:

\[ \text{type_spec} \{ \text{field} , \ldots \} \]

where

\[ \text{field ::= name} , \ldots : \text{expression} \]

Whenever a field has more than one name, it is equivalent to a sequence of fields, one for each name. Thus, if \text{R = record}[a: \text{Int}, b: \text{Int}, c: \text{Int}]. then the following two constructors are equivalent:
9.13.3 Structure, Record, and Atomic Record Constructors

\[ R(a, b: p(), c: 9) \]
\[ R(a: p(), b: p(), c: 9) \]

In the following we discuss only record constructors; structure and atomic record constructors are similar. In a record constructor, the type specification must name a record type: record \( S_1: T_1, \ldots, S_n: T_n \). This is the type of the constructed record. The component names in the field list must be exactly the names \( S_1, \ldots, S_n \), although these names may appear in any order. The expressions are evaluated left to right, and there is one evaluation per component name even if several component names are grouped with the same expression. The type of the expression for component \( S_i \) must be included in \( T_i \). The results of these evaluations form the components of a newly constructed record. This record is the value of the entire constructor expression.


Argus allows prefix and infix notation to be used as a shorthand for the operations listed in Table 9-1. The table shows the shorthand form and the computationally equivalent expanded form for each operation. For each operation, the type \( T \) is the type of the first operand.

<table>
<thead>
<tr>
<th>Shorthand form</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr₁ ** expr₂</td>
<td>T$power(expr₁, expr₂)</td>
</tr>
<tr>
<td>expr₁ // expr₂</td>
<td>T$mod(expr₁, expr₂)</td>
</tr>
<tr>
<td>expr₁ / expr₂</td>
<td>T$div(expr₁, expr₂)</td>
</tr>
<tr>
<td>expr₁ * expr₂</td>
<td>T$mul(expr₁, expr₂)</td>
</tr>
<tr>
<td>expr₁</td>
<td></td>
</tr>
<tr>
<td>expr₁ + expr₂</td>
<td>T$add(expr₁, expr₂)</td>
</tr>
<tr>
<td>expr₁ - expr₂</td>
<td>T$sub(expr₁, expr₂)</td>
</tr>
<tr>
<td>expr₁ &lt; expr₂</td>
<td>T$lt(expr₁, expr₂)</td>
</tr>
<tr>
<td>expr₁ &lt;= expr₂</td>
<td>T$le(expr₁, expr₂)</td>
</tr>
<tr>
<td>expr₁ = expr₂</td>
<td>T$equal(expr₁, expr₂)</td>
</tr>
<tr>
<td>expr₁ &gt;= expr₂</td>
<td>T$ge(expr₁, expr₂)</td>
</tr>
<tr>
<td>expr₁ &gt; expr₂</td>
<td>T$gt(expr₁, expr₂)</td>
</tr>
<tr>
<td>expr₁ ^= expr₂</td>
<td>- (expr₁ &lt; expr₂)</td>
</tr>
<tr>
<td>expr₁ ^= expr₂</td>
<td>- (expr₁ &lt;= expr₂)</td>
</tr>
<tr>
<td>expr₁ &gt;&gt;= expr₂</td>
<td>- (expr₁ = expr₂)</td>
</tr>
<tr>
<td>expr₁ &gt;&gt;= expr₂</td>
<td>- (expr₁ &gt;= expr₂)</td>
</tr>
<tr>
<td>expr₁ &gt;&gt;= expr₂</td>
<td>- (expr₁ &gt; expr₂)</td>
</tr>
<tr>
<td>expr₁ &amp; expr₂</td>
<td>T$and(expr₁, expr₂)</td>
</tr>
<tr>
<td>expr₁</td>
<td>expr₂</td>
</tr>
<tr>
<td>~ expr</td>
<td>T$not(expr)</td>
</tr>
</tbody>
</table>

Operator notation is used most heavily for the built-in types, but may be used for user-defined types as well. When these operations are provided for user-defined types, they should be free of side-effects, and
they should mean roughly the same thing as they do for the built-in types. For example, the comparison operations should only be used for types that have a natural partial or total order. Usually, the comparison operations \( (lt, le, equal, ge, gt) \) will be of type

\[
\text{proctype} \quad (T, T) \rightarrow \text{BooIl}
\]

the other binary operations (e.g., add, sub) will be of type

\[
\text{proctype} \quad (T, T) \rightarrow \text{T} \quad \text{signals} \quad (\ldots)
\]

and the unary operations will be of type

\[
\text{proctype} \quad (T) \rightarrow \text{T} \quad \text{signals} \quad (\ldots)
\]

9.15. Cand and Cor

Two additional binary operators are provided. These are the \textit{conditional and} operator, \texttt{cand}, and the \textit{conditional or} operator, \texttt{cor}. The result of evaluating:

\[
\text{expression}_1 \quad \text{cand} \quad \text{expression}_2
\]

is the boolean \texttt{and} of \texttt{expression}_1 and \texttt{expression}_2. However, if \texttt{expression}_1 is \texttt{false}, \texttt{expression}_2 is never evaluated. The result of evaluating:

\[
\text{expression}_1 \quad \text{cor} \quad \text{expression}_2
\]

is the boolean \texttt{or} of \texttt{expression}_1 and \texttt{expression}_2, but \texttt{expression}_2 is not evaluated unless \texttt{expression}_1 is \texttt{false}. For both \texttt{cand} and \texttt{cor}, \texttt{expression}_1 and \texttt{expression}_2 must have type \texttt{booI}.

Because of the conditional expression evaluation involved, uses of \texttt{cand} and \texttt{cor} are not equivalent to any procedure call.

9.16. Precedence

When an expression is not fully parenthesized, the proper nesting of subexpressions might be ambiguous. The following precedence rules are used to resolve such ambiguity. The precedence of each infix operator is given in the table below. Higher precedence operations are performed first. Prefix operators always have precedence over infix operators.

\[
\text{Table 9-2: Precedence for Infix Operators}
\]

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>++</td>
</tr>
<tr>
<td>4</td>
<td>* / //</td>
</tr>
<tr>
<td>3</td>
<td>+ -</td>
</tr>
<tr>
<td>2</td>
<td>&lt; &lt;= = =&gt; &gt; ~&lt; ~&lt;= ~= ~&gt; ~&gt;</td>
</tr>
<tr>
<td>1</td>
<td>&amp; cand</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
9.16 Precedence

The order of evaluation for operators of the same precedence is left to right, except for **, which is right to left.

9.17. Up and Down

There are no implicit type conversions in Argus. Two forms of expression exist for explicit conversions. These are:

up ( expression )
down ( expression )

Up and down may be used only within the body of a cluster operation (see Section 12.3). Up changes the type of the expression from the representation type of the cluster to the abstract type. Down converts the type of the expression from the abstract type to the representation type.
10. Statements

In this section, we discuss most of the statements of Argus, emphasizing the interaction of actions and the various kinds of control flow statements. We postpone discussion of the signal, emit, and except statements, which are used for signaling and handling exceptions, until Section 11. See Appendix I for the complete syntax of statements.

Atomic actions allow sequences of statements to appear to be indivisible to other actions. Sequences of statements that are not within an action are executed indivisibly; that is, other processes may observe intermediate states between statements. Statements are executed for their side-effects and do not return any values. Most statements are control statements; these permit the programmer to create processes and to dictate how control flows in a process. The rest are simple statements: assignment and calls (see Section 8).

A control statement can control a group of equates, declarations, and statements rather than just a single statement. Such a group is called a body, and has the form:

\[
\text{body} ::= \{ \text{equates} \} \\
\{ \text{statement} \}
\]

Note that statements include declarations (see Sections 7.1.2 and Appendix I). No special terminator is needed to signify the end of a body; semicolon marks used in the various compound statements serve to delimit the bodies. The statements in a body are executed sequentially in textual order.

10.1. Calls

A call statement may be used to call a procedure, handler, or creator. For procedures and handlers its form is the same as a call expression:

\[
\text{primary} ( [\text{expression}, \ldots ]) \\
\text{primary} ( [\text{expression}, \ldots ] \text{ @ primary })
\]

The primary must be a procedure, or handler object. The type of each actual expression must be included in the type of the corresponding formal argument. The procedure or handler may or may not return results; if it does return results, they are discarded.

For creator calls the syntax is similar, but one can optionally specify the node at which the guardian is to be created:

\[
\text{primary} ( [\text{expression}, \ldots ] \text{ @ primary })
\]

The primary following the at-sign (@) must be of type node.

The details of procedure, handler, and creator calls are described in Sections 8.2, 8.3, and 8.4.
10.2. Update Statements

Two special statements are provided for updating components of record and array-like objects. In addition they may be used with user-defined types with the appropriate properties. These statements resemble assignments syntactically, but are actually call statements.

10.2.1. Element Update

The element update statement has the form:

\[
\text{primary [ expression}_1 \text{ ] := expression}_2
\]

This form is merely syntactic sugar for a call of a \texttt{store} operation; it is equivalent to the call statement:

\[
T\texttt{store(primary, expression}_1, \text{expression}_2)
\]

where \( T \) is the type of the primary. \( T \) must provide a procedure named \texttt{store} that takes three arguments whose types include those of \texttt{primary, expression}_1, \text{and expression}_2, respectively.

10.2.2. Component Update

The component update statement has the form:

\[
\text{primary \ . name := expression}
\]

This form is syntactic sugar for a call of a \texttt{set} operation whose name is formed by attaching \texttt{set} to the name given. For example, if the name is \( f \), then the statement above is equivalent to the call statement:

\[
T\texttt{set}_f(\text{primary, expression})
\]

where \( T \) is the type of the primary. \( T \) must provide a procedure operation named \texttt{set}_f, where \( f \) is the name given in the component update statement. This procedure must take two arguments whose types include the types of \texttt{primary and expression}, respectively.

10.3. Block Statement

The block statement permits a sequence of statements to be grouped together into a single statement. Its form is:

\[
\text{begin body end}
\]

Since the syntax already permits bodies inside control statements, the main use of the block statement is to group statements together for use with the \texttt{except} statement (see Section 11).

10.4. Fork Statement

A fork statement creates an autonomous process. The fork statement has the form:

\[
\text{fork primary ( [ expression, ..., ] )}
\]

where the primary is a procedure object whose type has no results or signals (see Section 12.1). The type of each actual expression must be included in the type of the corresponding formal.

Execution of the fork statement starts by evaluating the primary and actual argument expressions from left to right. Any exceptions raised by the evaluation of the primary or the expressions are raised by the fork statement. If no exceptions are raised, then a new process is created and execution resumes after
the fork statement in the old process. The new process starts by calling the given procedure with the argument objects. This new process terminates if and when the procedure call does. However, if the guardian crashes the process goes away (like any other process).

Note that the new process does not run in an action, although the procedure called can start a topaction if desired. There is no mechanism for waiting for the termination of the new process. The procedure called in a fork statement cannot return any results or signal any exceptions.

10.5. Enter Statement
Sequential actions are created by means of the enter statement, which has two forms:

```
enter topaction body end
```

and

```
enter action body end
```

The topaction qualifier causes the body to execute as a new top-level action. The action qualifier causes the body to execute as a subaction of the current action; an attempt to execute an enter action statement in a process that is not executing an action is a programming error and causes the guardian to crash. When the body terminates, it does so either by committing or aborting. Normal completion of the body results in the action committing. Statements that transfer control out of the enter statement (exit, leave, break, continue, return, signal, and realsignal) normally commit the action unless are prefixed with abort (e.g., abort exit). Two-phase commit of a topaction may fail, in which case the enter topaction statement raises an unavailable exception.

10.6. Coenter Statement
Concurrent actions and processes are created by means of the coenter statement:

```
coenter coarm { coarm } end
```

where

```
coarm :::= armtag [ fooreach decl , ... in call ]
   body
```

```
armtag :::= action
  | topaction
  | process
```

Execution of the coenter starts by creating all of the coarm processes, sequentially, in textual order. A foreach clause indicates that multiple instances of the coarm will be created. The call in a foreach clause must be an iterator call. At each yield of the iterator, a new coarm process is created and the objects yielded are assigned to newly declared variables in that process. (This implicit assignment must be legal, see Section 6.1.) Each coarm process has separate, local instances of the variables declared in the foreach clause.
The process executing the \texttt{coenter} is suspended until after the \texttt{coenter} is finished. Once all coarm processes are created, they are started simultaneously as concurrent siblings. Each coarm instance runs in a separate process, and each coarm with an \texttt{armtag} of topanction or action executes within a new top-level action or subaction, respectively. An attempt to execute a \texttt{coenter} with a \texttt{process} coarm when in an action, or to execute a \texttt{coenter} with an \texttt{action} coarm when not in an action is an error and will cause the guardian to crash (see Table 10-1).

<table>
<thead>
<tr>
<th>\text{armtag}</th>
<th>\text{process executing the \texttt{coenter} is:}</th>
<th>\text{not in an action}</th>
<th>\text{running an action}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{action}</td>
<td>not legal</td>
<td>legal</td>
<td></td>
</tr>
<tr>
<td>\text{topaction}</td>
<td>legal</td>
<td>legal</td>
<td></td>
</tr>
<tr>
<td>\text{process}</td>
<td>legal</td>
<td>not legal</td>
<td></td>
</tr>
</tbody>
</table>

Table 10-1: Legality of \texttt{coenter} statements.

A simple example making use of \texttt{foreach} is:

\begin{verbatim}
coenter action foreach i: Int In Int$from_to (1, 5)
  p (i)
end
\end{verbatim}

which creates five processes, each with a local variable \(i\), having the value 1 in the first process, 2 in the second process, and so on. Each process runs in a newly created subaction. This statement is legal only if the process executing it is running an action.

A coarm may terminate without terminating the entire \texttt{coenter} (and sibling coarms) either by normal completion of its body, or by executing a \texttt{leave} statement (see Section 10.7). The commit of a coarm declared as a topanction may terminate in an \texttt{unavailable} exception if two-phase commit fails. Such an exception can only be handled outside the \texttt{coenter} statement, and thus will force termination of the entire \texttt{coenter} (as explained below).

A coarm may also terminate by transferring control outside the \texttt{coenter} statement. When such a transfer of control occurs, the following steps take place.

1. Any containing statements are terminated divisibly, to the outermost level of the coarm, at which point the coarm becomes the controlling coarm.
2. Once there is a controlling coarm, every other active coarm will be terminated (and abort if declared as an action) as soon as it leaves all \texttt{setze} statements; the controlling coarm is suspended until all other coarms terminate.
3. The controlling coarm then commits or aborts if declared as an action; if declared as a topanction and the two-phase commit fails, an \texttt{unavailable} exception is raised by the \texttt{coenter} statement.
4. Finally, the entire \texttt{coenter} terminates, and control flow continues outside the \texttt{coenter} statement.

Divisible termination implies, for instance, that a nested topanction may commit while its parent action aborts.
A simple example of early termination is reading from a replicated database, where any copy can supply the necessary information:

```coenter
coenter action foreach db: database in all_replicas (...)
  return( database$read (db))
end
```

When one of these coarms completes first, it tries to commit itself and abort the others. The aborts take place immediately (since there are no `seize` statements); it is not necessary for the handler calls to finish. It is possible that some descendants of an aborted coarm may be running at remote sites when the coarm aborts; the Argus system ensures that such orphans will be aborted before they can make their presence known or detect that they are in fact orphans (see Section 2.5).

### 10.7. Leave Statement

The leave statement has the form:

```coenter
[ abort ] leave
```

Executing a `leave` statement terminates the innermost `enter` statement or `coenter` coarm in which it appears. If the process terminated is an action, then it commits unless the `abort` qualifier is present, in which case the action aborts. The `abort` qualifier can only be used textually within an `enter` statement or within an action or `topaction` coarm of a `coenter` statement.

Note that unlike the other control flow statements, `leave` does not affect concurrent siblings in a `coenter` (see Section 10.6).

### 10.8. Return Statement

The form of the return statement is:

```coenter
[ abort ] return [ ( expression , ... ) ]
```

The return statement terminates execution of the containing routine. If the return statement occurs in an iterator no results can be returned. If the return statement is in a procedure, handler, or creator the type of each `expression` must be included in the corresponding return type of the routine. The expressions (if any) are evaluated from left to right, and the objects obtained become the results of the routine.

If no `abort` qualifier is present, then all containing actions (if any) terminated by this statement are committed. If the `abort` qualifier is present, then all terminated actions are aborted. Note that unlike the `leave` statement, `return` will abort concurrent siblings if executed within a coarm of a `coenter` statement (see Section 10.6). The `abort` qualifier can only be used textually within an `enter` statement, an action or `topaction` coarm of a `coenter` statement, or the body of a handler or creator.

Within a handler or creator, the result objects are encoded just before the activation action terminates, but after all control flow and nested action termination. If encoding of any result object terminates in a `failure` exception, then the activation action aborts and the handler or creator terminates with the same exception.
10.9. Yield Statement
The form of a yield statement is:

```
yield [( expression , ... )]
```

The yield statement may occur only in the body of an iterator. The effect of a yield statement is to suspend execution of the iterator invocation, and return control to the calling for statement or foreach clause. The values obtained by evaluating the expressions (left to right) are passed back to the caller. The type of each expression must be included in the corresponding yield type of the iterator. Upon resumption, execution of the iterator continues at the statement following the yield statement.

A yield statement cannot appear textually inside an enter, coenter, or setze statement.

10.10. Conditional Statement
The form of the conditional statement is:
```
If expression then body
   { elsif expression then body }
   [ else body ]
   end
```

The expressions must be of type bool. They are evaluated successively until one is found to be true. The body corresponding to the first true expression is executed, and the execution of the if statement then terminates. If there is an else clause and if none of the expressions is true, then the body in the else clause is executed.

10.11. While Statement
The while statement has the form:
```
while expression do body end
```

Its effect is to repeatedly execute the body as long as the expression remains true. The expression must be of type bool. If the value of the expression is true, the body is executed, and then the entire while statement is executed again. When the expression evaluates to false, execution of the while statement terminates.

10.12. For Statement
An iterator (see Section 12.2) can be called by a for statement. The iterator produces a sequence of items (where an item is a group of zero or more objects) one item at a time; the body of the for statement is executed for each item in the sequence.

The for statement has the form:
```
for [ decl , ... ] In call do body end
```
or
```
for [ idn , ... ] In call do body end
```
The call must be an iterator call. The second form (with an idn list) uses distinct, previously declared variables to serve as the loop variables, while the first form (with a decl list) form introduces new variables, local to the for statement, for this purpose. In either case, the type of each variable must include the corresponding yield type of the called iterator (see Section 12.2) and the number of variables must also match the yield type.

Execution of the for statement begins by calling the iterator, which either yields an item or terminates. If it yields an item (by executing a yield statement), its execution is temporarily suspended, the objects in the item are assigned to the loop variables, and the body of the for statement is executed. The next cycle of the loop is begun by resuming execution of the iterator after the yield statement which suspended it. Whenever the iterator terminates, the entire for statement terminates.

10.13. Break and Continue Statements

The break statement has the form:

\[
\text{[abort]} \text{break}
\]

Its effect is to terminate execution of the smallest for or while loop statement in which it appears. Execution continues with the statement following that loop.

The continue statement has the form:

\[
\text{[abort]} \text{continue}
\]

Its effect is to start the next cycle (if any) of the smallest for or while loop statement in which it appears.

Terminating a cycle of a loop may also terminate one or more containing actions. If no abort qualifier is present, then all these terminated actions (if any) are committed. If the abort qualifier is present, then all of the terminated actions are aborted. Unlike leave, break and continue will abort concurrent sibling actions when control flow leaves a containing coenter (see Section 10.6).

The abort qualifier can only be used textually within an enter statement or an action or topaction coarm of a coenter statement.

10.14. Tagcase Statement

The tagcase statement can be used to decompose oneof and variant objects; atomic_variant objects can be decomposed with the tagtest or tagwait statements. The decomposition is indivisible for variant objects; thus, use of the tagcase statement for variants is not equivalent to using a conditional statement in combination with is_ and value_ operations (see Section 11.15).

The form of the tagcase statement is:

\[
\text{tagcase expression}
\text{tag_arm \{ tag_arm \}}
\text{[ others : body ]}
\text{end}
\text{where}
\]

tag_arm ::= tag name, ... [( idn: type_spec )] : body

The expression must evaluate to a oneof or variant object. The tag of this object is then matched against the names on the tag Arms. When a match is found, if a declaration (idn: type_spec) exists, the value component of the object is assigned to the new local variable idn. The matching body is then executed; idn is defined only in that body. If no match is found, the body in the others arm is executed.

In a syntactically correct tagcase statement, the following three constraints are satisfied.
1. The type of the expression must be some oneof or variant type, T.
2. The tags named in the tag arms must be a subset of the tags of T, and no tag may occur more than once.
3. If all tags of T are present, there is no others arm; otherwise an others arm must be present.

On any tag_arm containing a declaration (idn: type_spec), type_spec must include the type(s) of T corresponding to the tag or tags named in that tag_arm.

10.15. Tagtest and Tagwait Statements

The tagtest and tagwait statements are provided for decomposing atomic_variant objects, permitting the selection of a body based on the tag of the object to be made indivisibly with the testing or acquisition of specified locks.

10.15.1. Tagtest Statement

The form of the tagtest statement is:

\[
\text{tagtest expression}
\]

\[
\text{atag arm \{ atag arm \} }
\]

\[
\text{[ others : body ]}
\]

where

\[
\text{atag arm ::= tag kind name, ... [( idn: type_spec )] : body}
\]

\[
tag kind ::= tag
\]

\[
\mid \text{wtag}
\]

The expression must evaluate to an atomic_variant object. If a read lock could be obtained on the atomic_variant object by the current action, then the tag of the object is matched against the names on the atag arms; otherwise the others arm, if present, is executed. If a matching name is found, then the tag kind is considered.

- If the tag kind is tag, a read lock is obtained on the object and the match is complete.
- If the tag kind is wtag and the current action can obtain a write lock on the object, then a write lock is obtained and the match is complete.

When a complete match is found, if a declaration (idn: type_spec) exists, the value component of the object is assigned to the new local variable idn. The matching body is then executed; idn is defined only in that body. The entire matching process, including testing and acquisition of locks, is indivisible.
10.15.1 Tagtest Statement

If a complete match is not found, or the object was not readable by the action, then the others arm (if any) is executed; if there is no others arm, the tagtest statement terminates. If no complete match is found, then no locks are acquired.

The tagtest statement will only obtain a lock if it is possible to do so without "waiting". For example, suppose that the internal state of the atomic_variant indicates that some previous action acquired a conflicting lock. This action may have since aborted, or may have committed up to an ancestor of the action executing the tagtest, but determining such facts may require system-level communication to other guardians. In this case the tagtest statement may give misleading information, because it may not indicate a match. Apparent anomalies in testing locks may occur even if the action executing the tagtest "knows" that the lock can be acquired, so that the use of tagtest to avoid deadlocks or long delays may result in excessive aborts.

10.15.2. Tagwait Statement

The form of the tagwait statement is:

```
tagwait expression
   atag_arm { atag_arm }
end
```

Execution of the tagwait statement proceeds as for the tagtest statement, but if no complete match is found, or if the object is not readable by the current action, then the entire matching process is repeated (after a system-controlled delay), until a complete match is found. Although there is no others arm in a tagwait statement, all tag names do not have to be listed.

10.15.3. Common Constraints

Tagtest and tagwait statements may be executed only within an action. An attempt to execute a tagtest or tagwait statement in a process that is not executing an action is an error and will cause the guardian to crash after evaluating the expression.

In a syntactically correct tagtest or tagwait statement, the following three constraints are satisfied.

1. The type of the expression must be some atomic_variant type, T.

2. The tags named in the atag_arms must be a subset of the tags of T, and no tag may occur more than once.

3. Finally, on any atag_arm containing a declaration (idn: type_spec), type_spec must include the type(s) specified as corresponding in T to the tag or tags named in the atag_arm.

A simple example of a tagtest statement is garbage collecting the elements of an array that are in the dequeued state:
item = atomic_value(enqueued: int, dequeued: null)
for i in array[items][Statements] do
  tag these
  tag dequeued: array[items][from(i)]
  object: break
  end
end

10.16. Seize Statement

The seize statement has the form:

```
seize expression via body end
```

The expression must evaluate to a mutex object. The enclosing process then attempts to gain possession of that mutex object, and walks to do so if successful. Only one process, whether user or system defined, may possess a given mutex object at one time. Once the process gains possession, the body of the seize statement is executed. When the body completes, possession of the mutex is released. This includes termination of the body by statements that terminate control out of the body.

The body of a seize statement is considered to be a critical section; a process executing in the body of a seize statement can only be feasibly terminated by causing the process at which the process is running. See Section 10 for the reasons for this and for more detailed semantics of mutex.

Multiple, nested seizures of the same mutex object are allowed, and must proceed. A process seizing a mutex that it has already seized will not deadlock with itself, and possession is not really released until the outermost seize terminates.

10.17. Pause Statement

The pause statement has the form:

```
pause
```

The pause statement must occur within an enclosing seize statement. Its effect is to release the mutex object associated with the smallest enclosing seize statement, suspend execution of the process for a system-controlled period of time, and then reinitialize possession and continue execution.

If multiple, nested seizures on the mutex object have been performed, pause will not actually release possession. For example, possession is not released in the following:

```
seize m do
  seize m do
    pause       % does not really release possession
    end
  end
end
```

In general, nested seizures should be avoided when pause must be used, and pause should be avoided when nested seizures must be used.
10.18. Terminate Statement

The terminate statement may occur only within a guardian definition (see Sect 13). The form of a
terminate statement is:

\texttt{terminate}

When executed within an action, its effect is to cause the eventual destruction of the guardian after the
enclosing action commits to the top. If a process attempts to execute terminate while not running an
action, a topaction is created to execute the terminate and immediately commit.

Let \( A \) be the action that is executing the terminate. The effect of this statement is the following:

1. Action \( A \) must wait until the action that created the guardian is committed relative to \( A \). In
   the case of a permanent guardian whose creation has committed to the top there will be no
   wait, but for a recently created guardian there may be a delay.

2. If multiple processes are attempting to execute terminate statements, at most one at a
   time may proceed to the next step.

3. If \( A \) commits to the top, the guardian will be destroyed at some time after topaction commit.
   If some ancestor of \( A \) aborts, however, the guardian will be unaffected. The guardian is
   also unaffected during the time between \( A \) executing terminate and \( A \) committing to the
   top.

In order to avoid serialization problems, creation or destruction of a guardian must be synchronized
with use of that guardian via atomic objects such as the catalog (see Section 3.4).
11. Exception Handling and Exits

A routine is designed to perform a certain task. However, in some cases that task may be impossible to perform. In such a case, instead of returning normally (which would imply successful performance of the intended task), the routine should notify its caller by signalling an exception, consisting of a descriptive name and zero or more result objects.

The exception handling mechanism consists of two parts: signalling exceptions and handling exceptions. Signalling is the way a routine notifies its caller of an exceptional condition; handling is the way the caller responds to such notification. A signalled exception always goes to the immediate caller, and the exception must be handled in that caller. When a routine signals an exception, the current activation of that routine terminates and the corresponding call (in the caller) is said to raise the exception. When a call raises an exception, control immediately transfers to the closest applicable exception handler. Exception handlers are attached to statements; when execution of the exception handler completes, control passes to the statement following the one to which the exception handler is attached. For brevity, exception handlers will be called "handlers" in this chapter; these should not be confused with the remote call handlers of guardians (see Section 13).

11.1. Signal Statement

An exception is signalled with a signal statement, which has the form:

```
[ abort ] signal name [ ( expression , ... ) ]
```

A signal statement may appear anywhere in the body of a routine. The execution of a signal statement begins with evaluation of the expressions (if any), from left to right, to produce a list of exception results. The activation of the routine is then terminated. Execution continues in the caller as described in Section 11.2 below.

The exception name must be one of the exception names listed in the routine heading. If the corresponding exception specification in the heading has the form:

```
name(T_1, ..., T_n)
```

then there must be exactly $n$ expressions in the signal statement, and the type of the $i$th expression must be included in $T_i$.

If no abort qualifier is present, then all containing actions (if any) terminated by this statement are committed. If the abort qualifier is present, then all terminated actions are aborted. Unlike the leave statement, signal will terminate (abort) concurrent siblings if executed within a coenter statement (see Section 10.6). The abort qualifier can only be used textually within an enter statement, an action or topaction coarm of a coenter statement, or the body of a handler or creator.

Within a handler or creator, the result objects are encoded just before the activation action terminates, but after termination of all control flow and nested actions. If encoding of any result object terminates in a failure exception, then the activation action aborts and the handler or creator terminates with the failure exception.
11.2. Except Statement

When a routine activation terminates by signalling an exception, the called routine is said to raise that exception. By attaching exception handlers to statements, the caller can specify the action to be taken when an exception is raised by a call within a statement or by the statement itself.

A statement with handlers attached is called an except statement, and has the form:

```
statement except { when_handler }
    [ others_handler ]
end
```

where

- `when_handler ::= when name , ... [ ( decl , ... ) ] : body`
- `when name , ... ( * ) : body`

- `others_handler ::= others [ ( idn : string ) ] : body`

Let $S$ be the statement to which the handlers are attached, and let $X$ be the entire except statement. Each `when_handler` specifies one or more exception names and a `body`. The `body` is executed if an exception with one of those names is raised by a call in $S$. Each of the names listed in the `when_handlers` must be distinct. The optional `others_handler` is used to handle all exceptions not explicitly named in the `when_handlers`. The statement $S$ can be any form of statement, and can even be another except statement. As an example, consider the following except statement:

```
m.send_mail(user, my_message)
    except when no_such_user: ... % body 1
    when unavailable, failure (s: string): ... % body 2
    when others (ename: string): ... % body 3
end
```

This statement handles exceptions arising from a remote call. If the call raises a `no_such_user` exception, then "body 1" will be executed. If the call raises a `failure` or `unavailable` exception, then "body 2" will be executed. Any other exception will be handled by "body 3."

If, during the execution of $S$, some call in $S$ raises an exception $E$, control transfers to the textually closest handler for $E$ that is attached to a statement containing the call. When execution of the handler completes, control passes to the statement following the one to which the handler is attached. Thus if the closest handler is attached to $S$, the statement following $X$ is executed next. If execution of $S$ completes without raising an exception, the attached handlers are not executed.

An exception raised inside a handler is treated the same as any other exception: control passes to the closest handler for that exception. Note that an exception raised in some handler attached to $S$ cannot be handled by any handler attached to $S$; the exception can be handled within the handler, or it can be handled by some handler attached to a statement containing $X$. For example, in the following except statement:
times3_plus1(a)
except when limits:
  a := a + a
  when overflow: ... % body 2
end

any overflow signal raised by the expression \( a + a \) will not be handled in "body 2," because this overflow handler is not in an except statement attached to the assignment statement \( a := a + a \).

We now consider the forms of exception handlers in more detail. The form:

\[
\text{when name, ... [ ( decl, ... ) ] : body}
\]

is used to handle exceptions with the given names when the exception results are of interest. The optional declared variables, which are local to the handler, are assigned the exception results before the body is executed. Every exception potentially handled by this form must have the same number of results as there are declared variables, and the types of the variables must include the types of the results. The form:

\[
\text{when name, ... ( *) : body}
\]

handles all exceptions with the given names, regardless of whether or not there are exception results; any actual results are discarded. Using this form, exceptions with differing numbers and types of results can be handled together.

The form:

\[
\text{others [ ( idn : string ) ] : body}
\]

is optional, and must appear last in a handler list. This form handles any exception not handled by other handlers in the list. If a variable is declared, it must be of type string. The variable, which is local to the handler, is assigned a lower case string representing the actual exception name; any results are discarded.

Note that number and type of exception results are ignored when matching exceptions to handlers; only the names of exceptions are used. Thus the following is illegal, in that Int$div signals zero_divide without any results (see Section II.4), but the closest handler has a declared variable:

\[
\begin{align*}
\text{begin} \\
\quad y : \text{int} := 0 \\
\quad x : \text{int} := 3 / y \\
\quad \text{except when zero_divide (z: int): return end} \\
\end{align*}
\]

A call need not be surrounded by except statements that handle all potential exceptions. In many cases the programmer can prove that a particular exception will not arise; for example, the call Int$div(x, 7) will never signal zero_divide. However, if some call raises an exception for which there is no handler, then the guardian crashes due to this error\(^9\).

\(^9\)The implementation of the Argus should log unhandled exceptions in some fashion, to aid later debugging. During debugging, an unhandled exception would be trapped by the debugger before the crash.
11.3. Resignal Statement

A resignal statement is a syntactically abbreviated form of exception handling:

statement [ abort ] resignal name, ...

Each name listed must be distinct, and each must be one of the condition names listed in the routine heading. The resignal statement acts like an except statement containing a handler for each condition named, where each handler simply signals that exception with exactly the same results. Thus, if the resignal clause names an exception with a specification in the routine heading of the form:

name(T₁, ..., Tₙ)

then effectively there is a handler of the form:

when name (x₁: T₁, ..., xₙ: Tₙ): [ abort ] signal name(x₁, ..., xₙ)

which has an abort qualifier if and only if the resignal statement did. As for an explicit handler of this form, every exception potentially handled by this implicit handler must have the same number of results as declared in the exception specification, and the types of the results must be included in the types listed in the exception specification.

If no abort qualifier is present, then all containing actions (if any) terminated by this statement are committed. If the abort qualifier is present, then all terminated actions are aborted. Unlike the leave statement, resignal will abort concurrent siblings if executed within a coenter statement (see Section 10.6). The abort qualifier can only be used textually within an enter statement, an action or topaction coarm of a coenter statement, or the body of a handler or creator.

11.4. Exit Statement

An exit statement has the form:

[ abort ] exit name [( expression, ... )]

An exit statement is similar to a signal statement except that where the signal statement signals an exception to the calling routine, the exit statement raises the exception directly in the current routine. Thus an exit causes a transfer of control within a routine but does not terminate the routine. An exception raised by an exit statement must be handled explicitly by a containing except statement with a handler of the form:

when name, ... [( decl, ... )]: body

As usual, the types of the expressions in the exit statement must be included in the types of the variables declared in the handler. The handler must be an explicit one, i.e., exits to the implicit handlers of resignal statements are illegal.

If no abort qualifier is present, then all containing actions (if any) terminated by the exit statement are committed. If the abort qualifier is present, then all terminated actions are aborted. Unlike the leave statement, exit will abort concurrent siblings when control flow leaves a containing coenter statement (see Section 10.6). The abort qualifier can only be used textually within an enter statement or an action or topaction coarm of a coenter statement.
The exit statement and the signal statement mesh nicely to form a uniform mechanism. The signal statement can be viewed simply as terminating a routine activation; an exit is then performed at the point of invocation in the caller. (Because this exit is implicit, it is not subject to the restrictions on exits listed above.)

11.5. Exceptions and Actions

A new action is created by a handler call, creator call, enter statement, or action or topaction arm of a coenter statement. In addition, the recover code of a guardian runs as an action. When control flows out of an action, that action is committed unless action is taken to prevent its committing. To abort an action, it is necessary to qualify control flow statements such as exit, signal, resignal, and leave with the keyword abort (see Section 10).

However, there is an additional complication. Not only will explicit termination of actions by exit, signal, and resignal statements commit actions, but also implicit termination by flow of control out of an action body when an exception raised within that body is handled outside the action's body. Thus, if an exception which is raised by a call within an action is not to commit the action, then it is necessary to catch the exception within the action. This is particularly important when dealing with topactions. A common desire is to catch all "unexpected" exceptions, but still have the topaction abort. In this case, the catch-all exception handler must be placed inside the topaction. However, an unavailable handler must still be placed outside the topaction, since the two-phase commit may fail.

An action or topaction coarm of a coenter statement will not abort its concurrent siblings when it ends in either normal completion of its body or by a leave statement. However, if control flows otherwise out of the coenter statement from within one of the coarms, the entire coenter is terminated as described in Section 10.6. Thus, a coenter statement should be used carefully to ensure the proper behavior in case of exceptions. There may be circumstances where a separate exception handler will have to be used for each coarm to ensure the proper behavior, even when the exception handling is identical for each coarm.

11.6. Failure Exceptions

Argus responds to unhandled exceptions differently than CLU. In CLU, an unhandled exception in some routine causes that routine to terminate with the failure exception. In Argus, however, an unhandled exception causes the guardian that is running the routine to crash. Our motivation for this change is that an unhandled exception is typically a symptom of a programming error that cannot be handled by the calling routine. Furthermore, crashing the guardian limits the damage that the programming error can cause.

Procedures and iterators in Argus no longer have an implicit failure exception associated with them. Instead, such a routine may list failure explicitly in its signals clause and failure may have any number (and type) of exception results. Failure should be used to indicate an unexpected (and possibly
catastrophic) failure of a lower-level abstraction, for example, when there is a failure in a type parameter's routines (for instance in similar or copy operations). Another example is when there is an unwanted side effect, such as a bounds exception in array[1] elements caused by a mutation of the array argument. Various operations of the built-in types signal failure under such circumstances.

For handlers and creators, failure is used to indicate that a remote call has failed; thus the exception failure(string) is implicit in the type of every handler and creator (see Section 13.5). When a remote call terminates with the failure exception, this means that not only has this call failed, but that the call is unlikely to succeed if repeated.
12. Modules

Besides guardian modules, Argus has procedures, iterator, cluster, and equate modules.

\[
\text{module ::= \{ equate \} guardian} \\
\quad \{ \text{equate} \} \text{ procedure} \\
\quad \{ \text{equate} \} \text{ iterator} \\
\quad \{ \text{equate} \} \text{ cluster} \\
\quad \{ \text{equate} \} \text{ equates}
\]

Guardians are discussed in Section 13, the rest are described below.

12.1. Procedures

A procedure performs an action on zero or more arguments, and when it terminates it returns zero or more results. A procedure implements a procedural abstraction: a mapping from a set of argument objects to a set of result objects, with possible modification of some of the argument objects. A procedure may terminate in one of a number of conditions; one of these is the normal condition, while others are exceptional conditions. Differing numbers and types of results may be returned in the different conditions.

The form of a procedure is:

\[
\text{idn = proc \{ \text{parms} \} \text{argss} \{ \text{returns} \} \{ \text{signals} \} \{ \text{where} \} \\
\quad \text{routine\_body} \\
\quad \text{end idn}
\]

where

\[
\begin{align*}
\text{args} & \quad ::= ( [ \text{decl} , \text{...} ] ) \\
\text{returns} & \quad ::= \text{returns} ( \text{type\_spec} , \text{...} ) \\
\text{signals} & \quad ::= \text{signals} ( \text{exception} , \text{...} ) \\
\text{exception} & \quad ::= \text{name} [ ( \text{type\_spec} , \text{...} ) ] \\
\text{routine\_body} & \quad ::= \{ \text{equate} \} \\
\quad & \quad \{ \text{own\_var} \} \\
\quad & \quad \{ \text{statement} \}
\end{align*}
\]

In this section we discuss non-parameterized procedures, in which the \text{parms} and \text{where} clauses are missing. Parameterized modules are discussed in Section 12.5. Own variables are discussed in Section 12.7.

The heading of a procedure describes the way in which the procedure communicates with its caller. The \text{args} clause specifies the number, order, and types of arguments required to call the procedure, while the \text{returns} clause specifies the number, order, and types of results returned when the procedure terminates normally (by executing a return statement or reaching the end of its body). A missing \text{returns} clause indicates that no results are returned.

The \text{signals} clause names the exceptional conditions in which the procedure can terminate, and specifies the number, order, and types of result objects returned in each condition. All names of
exceptions in the signals clause must be distinct. The idn following the end of the procedure must be the same as the idn naming the procedure.

A procedure is an object of some procedure type. For a non-parameterized procedure, this type is derived from the procedure heading by removing the procedure name, rewriting the formal argument declarations with one idn per decl, deleting the idns of all formal arguments, and finally, replacing proc by proctype.

The call of a procedure causes the introduction of the formal variables, and the actual arguments are assigned to these variables. Then the procedure body is executed. Execution terminates when a return statement or a signal statement is executed, or when the textual end of the body is reached. If a procedure that should return results reaches the textual end of the body, the guardian crashes due to this error. At termination the result objects, if any, are passed back to the caller of the procedure.

12.2. Iterators

An iterator computes a sequence of items, one item at a time, where an item is a group of zero or more objects. In the generation of such a sequence, the computation of each item of the sequence is usually controlled by information about what previous items have been produced. Such information and the way it controls the production of items is local to the iterator. The user of the iterator is not concerned with how the items are produced, but simply uses them (through a for statement) as they are produced. Thus the iterator abstracts from the details of how the production of the items is controlled; for this reason, we consider an iterator to implement a control abstraction. Iterators are particularly useful as operations of data abstractions that are collections of objects (e.g., sets), since they may produce the objects in a collection without revealing how the collection is represented.

An iterator has the form:

\[ \text{idn} = \text{iter} \left[ \text{parms} \right] \text{args} \left[ \text{yields} \right] \left[ \text{signals} \right] \left[ \text{where} \right] \text{routine_body} \text{end idn} \]

where

\[ \text{yields ::= yields (type_spec,...)} \]

In this section we discuss non-parameterized iterators, in which the parms and where clauses are missing. Parameterized modules are discussed in Section 12.5. Own variables are discussed in Section 12.7.

The form of an iterator is similar to the form of a procedure. There are only two differences:

1. An iterator has a yields clause in its heading in place of the returns clause of a procedure. The yields clause specifies the number, order, and types of objects yielded each time the iterator produces the next item in the sequence. If zero objects are yielded, then the yields clause is omitted. The idn following the end of the iterator must be the same as the idn naming the iterator.

2. Within the iterator body, the yield statement is used to present the caller with the next item
in the sequence. An iterator terminates in the same manner as a procedure, but it may not
return any results.

An iterator is an object of some iterator type. For a non-parameterized iterator, this type is derived from
the iterator heading by removing the iterator name, rewriting the formal argument declarations with one
idn per dec1, deleting the idns of all formal arguments, and finally, replacing lit by litertype.

An iterator can be called only by a for statement or by a foreach clause in a coenter statement.

12.3. Clusters

A cluster is used to implement a new data type, distinct from any other built-in or user-defined data
type. A data type (or data abstraction) consists of a set of objects and a set of primitive operations. The
primitive operations provide the most basic ways of manipulating the objects; ultimately every
computation that can be performed on the objects must be expressed in terms of the primitive operations.
Thus the primitive operations define the lowest level of observable object behavior10.

The form of a cluster is:

\[ \text{idn = cluster [ parms ] is opidn, ..., [ where ]} \]
\[ \text{cluster\_body} \]
\[ \text{end idn} \]

where

\[ \text{opidn ::= idn} \]
\[ \text{transmit} \]

\[ \text{cluster\_body ::= \{ equate \} rep = type\_spec \{ equate \}} \]
\[ \{ own\_var \} \]
\[ \text{routine \{ routine \}} \]

\[ \text{routine ::= procedure} \]
\[ \text{iterator} \]

In this section we discuss non-parameterized clusters, in which the parms and where clauses are
missing. Parameterized modules are discussed in Section 12.5. Own variables are discussed in Section
12.7.

The primitive operations are named by the list of opidns following the reserved word is. All of the
opidns in this list must be distinct. The idn following the end of the cluster must be the same as the idn
naming the cluster.

To define a new data type, it is necessary to choose a concrete representation for the objects of the
type. The special equate:

---

10Readers not familiar with the concept of data abstraction might read Liakov, B. and Guttag, J., Abstraction and Specification in
rep = type_spec
within the cluster body identifies the type_spec as the concrete representation. Within the cluster, rep
may be used as an abbreviation for this type_spec.

The identifier naming the cluster is available for use in the cluster body. Use of this identifier within the
cluster body permits the definition of recursive types.

In addition to giving the representation of objects, the cluster must implement the primitive operations
of the type. One exception to this, however, is the transmit operation. The transmit operation is not
directly implemented by a cluster; instead, the cluster must implement two operations: encode and
decode (see Section 14 for details). The primitive operations may be either procedural or control
abstractions; they are implemented by procedures and iterators, respectively. Any additional routines
implemented within the cluster are hidden: they are private to the cluster and may not be named directly
by users of the abstract type. All the routines must be named by distinct identifiers; the scope of these
identifiers is the entire cluster.

Outside the cluster, the type's objects may only be treated abstractly (i.e., manipulated by using the
primitive operations). To implement the operations, however, it is usually necessary to manipulate the
objects in terms of their concrete representation. It is also convenient sometimes to manipulate the
objects abstractly. Therefore, inside the cluster it is possible to view the type's objects either abstractly or
in terms of their representation. The syntax is defined to specify unambiguously, for each variable that
refers to one of the type's objects, which view is being taken. Thus, inside a cluster named T, a
declaration:

\[ v : T \]
indicates that the object referred to by v is to be treated abstractly, while a declaration:

\[ w : \text{rep} \]
indicates that the object referred to by w is to be treated concretely. Two primitives, up and down, are
available for converting between these two points of view. The use of up permits a type rep object to be
viewed abstractly, while down permits an abstract object to be viewed concretely. For example, given
the declarations above, the following two assignments are legal:

\[ v := \text{up}(w) \]
\[ w := \text{down}(v) \]
Only routines inside a cluster may use up and down. Note that up and down are used merely to inform
the compiler that the object is going to be viewed abstractly or concretely, respectively.

A common place where the view of an object changes is at the interface to one of the type's
operations: the user, of course, views the object abstractly, while inside the operation, the object is
viewed concretely. To facilitate this usage, a special type specification, cvt, is provided. The use of cvt
is restricted to the args, returns, yields and signals clauses of routines inside a cluster, and may be used
at the top level only (e.g., \text{array[cvt]} is illegal). When used inside the args clause, it means that the view
of the argument object changes from abstract to concrete when it is assigned to the formal argument
variable. When cvt is used in the returns, yields, or signals clause, it means the view of the result object
changes from concrete to abstract as it is returned (or yielded) to the caller. Thus cvt means abstract outside, concrete inside: when constructing the type of a routine, cvt is equivalent to the abstract type, but when type-checking the body of a routine, cvt is equivalent to the representation type. The type of each routine is derived from its heading in the usual manner, except that each occurrence of cvt is replaced by the abstract type. The cvt form does not introduce any new ability over what is provided by up and down. It is merely a shorthand for a common case.

Inside the cluster, it is not necessary to use the compound form (type_spec$op_name) for naming locally defined routines. Furthermore, the compound form cannot be used for calling hidden routines.

12.4. Equate Modules

An equate module provides a convenient way to define a a set of equates for later use by other modules.

The form of an equate module is:

```
iden = equates [ parms [ where ] ]
    equate { equate }
end iden
```

The usual scope rules apply. The iden following the end of the equate module must be the same as the iden naming the equate module.

In this section we discuss non-parameterized equate modules. Parameterized modules are discussed in Section 12.5.

An equate module defines a set of equates, that is, it defines a set of named constants. The set of equates is also a constant, although it is not an object. Thus the name of an equate module can be used in an equate, but an equate module cannot be assigned to a variable. The equates defined by an equate module E may be referenced using the same syntax as for naming the operations of a cluster. For example, an object or type named n in equate module E can be referred to as E$n$. If equate modules contain equates that give names to other equate modules, compound names can be used. For example:

```
A[int]$B$C$name
```

where A, B, and C are equate modules is legal.

As always, equates to type specifications do not define new types but merely abbreviations for types. For example, in the following:

```
my_types = equates
    a1 = array[int]
    float = real
end my_types
```

the types my_types$a1 and array[int] are equivalent.
12.5. Parameterized Modules

Procedures, iterators, clusters, guardians (see Section 13), and equate modules may all be parameterized. Parameterization permits a set of related abstractions to be defined by a single module. In each module heading there is an optional \texttt{parms} clause and an optional \texttt{where} clause (see Appendix I). The presence of the \texttt{parms} clause indicates that the module is parameterized; the \texttt{where} clause declares the types of any operation parameters that are expected to accompany the formal type parameters.

The form of the \texttt{parms} clause is:

\[
[ \texttt{parm} , \ldots ]
\]

where

\[
\texttt{parm} ::= \texttt{idn} , \ldots : \texttt{type} \quad \text{spec}
\]

Each \texttt{parm} declares some number of formal parameters. Only the following types of parameters can be declared in a \texttt{parms} clause: \texttt{int}, \texttt{real}, \texttt{bool}, \texttt{char}, \texttt{string}, \texttt{null}, and \texttt{type}. The declaration of operation parameters associated with type parameters is done in the \texttt{where} clause, as discussed below. The actual values for parameters are required to be constants that can be computed at compile-time. This requirement ensures that all types are known at compile-time, and permits complete compile-time type-checking.

In a parameterized module, the scope rules permit the parameters to be used throughout the module. Type parameters can be used freely as type specifications, and all other parameters (including the operations parameters specified in the \texttt{where} clause) can be used freely as expressions.

A parameterized module implements a set of related abstractions. A program must \texttt{instantiate} a parameterized module before it can be used; that is, it must provide actual, constant values for the parameters (see Section 12.6). The result of an instantiation is a procedure, iterator, type, guardian, or equate module that may be used just like a non-parameterized module of the same kind. Each distinct list of actual parameters produces a distinct procedure, iterator, type, guardian, or equate module (see Section 12.6 for details).

The meaning of a parameterized module is given by binding the actual parameters to the formal parameter names and deleting the \texttt{parms} clause and the \texttt{where} clause. That is, in an an instantiation of a parameterized module, each formal parameter name denotes the corresponding actual parameter. The resulting module is a regular (non-parameterized) module. In the case of a cluster some of the operations may have additional parameters; further bindings take place when these operations are instantiated.

In the case of a type parameter, one can also declare what operation parameters must accompany the type by using a \texttt{where} clause. The \texttt{where} clause also specifies the type of each required operation parameter. The \texttt{where} clause constrains the parameterized module as well: the only operations of the type parameter that can be used are those listed in the \texttt{where} clause.
12.5 Parameterized Modules

The form of the where clause is:

\[
\text{where} ::= \text{where restriction, ...}
\]

\[
\text{restriction} ::= \text{idn has oper_decl, ...}
\]

\[
| \quad \text{idn in type_set}
\]

\[
\text{oper_decl} ::= \text{name, ... : type_spec}
\]

\[
| \quad \text{transmit}
\]

\[
\text{type_set} ::= \{ \text{idn} \mid \text{idn has oper_decl, ...} \{ \text{equate} \} \}
\]

\[
| \quad \text{idn}
\]

\[
| \quad \text{reference} \$ \text{name}
\]

There are two forms of restrictions. In both forms, the initial \text{idn} must be a type parameter. The \text{has} form lists the set of required operation parameters directly, by means of \text{oper_decl}s. The \text{type_spec} in each \text{oper_decl} must be a proctype, lertype, or creartotype (see Appendix I). The \text{in} form requires that the actual type be a member of a \text{type_set}, a set of types with the required operations. The two identifiers in the \text{type_set} must match, and the notation is read like set notation; for example,

\[
\{ t \mid t \text{ has } f : \ldots \}
\]

means "the set of all types \text{t} such that \text{t has } f : \ldots". The scope of the identifier is the \text{type_set}.

The \text{in} form is useful because an abbreviation can be given for a \text{type_set} via an equate. If it is helpful to introduce some abbreviations in defining the \text{type_set}, these are given in the optional equates within the \text{type_set}. The scope of these equates is the entire \text{type_set}.

A routine in a parameterized cluster may have a where clause in its heading, and can place further constraints on the cluster parameters. For example, any \text{type} is permissible for the array element type, but the array \text{similar} operation requires that the element type have a \text{similar} operation. This means that \text{array}[\text{T}] exists for any \text{T}, but that \text{array}[\text{T}]\text{similar} exists only when an actual operation parameter is provided for \text{T}\text{similar} (see Section 12.6). Note that a routine need not include in its where clause any of the restrictions included in the cluster where clause.

12.6. Instantiations

To instantiate a parameterized module, constants or type specifications are provided as actual parameters:

\[
\text{actual_parm} ::= \text{constant}
\]

\[
| \quad \text{type_actual}
\]

\[
\text{type_actual} ::= \text{type_spec \{ with \{ opbinding, \ldots \} \}}
\]

\[
\text{opbinding} ::= \text{name, \ldots : primary}
\]

If the parameter is a \text{type}, the module's where clause may require that some routines be passed as parameters. These routines can be passed implicitly by omitting the with clause; the routine selected as a default will be the operation of the type that has the same name as that used in the where clause.
Routines may also be passed explicitly by using the with clause, overriding the default. In this case, the actual routine parameter need not have the same name as is required in the where clause, and need not even be one of the type’s primitive operations.

The syntactic sugar that allows default routines to be selected implicitly works as follows. If a generator requires an operation named op from a type parameter, and if the corresponding type_actual, TS with {...}, has no explicit binding for op, then Argus adds an opbinding of op to TS$op. (It will be an error if TS$op is not defined.) Thus one only has to provide an explicit opbinding if the default is unsatisfactory.

For example, suppose a procedure generator named sort has the following heading:

\[
\text{sort = proc(t:: type)(a:: array[t]) where t has gt: proctype(t,t) returns(bool)}
\]

and consider the three instantiations:

\[
\begin{align*}
\text{sort[int with {gt: Int$gt}]} & \\
\text{sort[int]} & \\
\text{sort[int with {lt: Int$lt}]} & 
\end{align*}
\]

The first two instantiations are equivalent; in the first the routine Int$gt is passed explicitly, while in the second it is passed implicitly as the default. In the third instantation, however, Int$lt is passed in place of the default. All three instantiations result in a routine of type:

\[
\text{proctype (array[Int])}
\]

and so each could be called by passing it an array[Int] as an argument. However a call of the third instantiation will sort its array argument in the opposite order from a call of either the first or second instantiation.

Within an instantiation of a parameterized module, an operation of a type parameter named $op denotes the actual routine parameter bound to op in the instantiation of that module. For example, suppose we make the call:

\[
\text{sort[lt with {gt: Int$lt}]} (my_ints)
\]

where my_ints is an array of integers. If, in the body of sort, there is a recursive call:

\[
\text{sort[lt with {gt: $gt}]} (a, i, j)
\]

then t denotes the type Int, and $gt denotes the routine Int$lt, so that the recursive sort happens in the correct order.

A cluster generator may include routines with where clauses that place additional requirements on the cluster’s type parameters. A common example is to require a copy operation only within the cluster’s copy implementation.

\[
\begin{align*}
\text{set = cluster[t:: type] is ...; copy} & \\
\text{where t has equal: proctype(t,t) returns(bool)} & \\
\text{rep = array[t]} & \\
\text{...} & \\
\text{copy = proc(s:: cvt) returns(cvt) where t has copy: proctype(t) returns(t)} & \\
\text{return(rep$copy(s))} & \\
\text{end copy}
\end{align*}
\]

The intent of these subordinate where clauses is to allow more operations to be defined if the actual type parameter has the additional required operations, but not to make the additional operations an absolute
requirement for obtaining an instance of the type generator. For example, with the above definition of set, 
set[any] would be defined, but set[any]$copy would not be defined because any does not have a copy 
operation. We shall call the routine parameters required by subordinate where clauses optional 
parameters.

Like regular required parameters, optional parameters can be provided when the cluster as a whole is 
instantiated and can be provided explicitly or by default. For any optional parameter op that is not 
provided explicitly by the type_actual, TS with { ... }, we add an opbinding of op to TS$op if TS$op exists; 
otherwise the opbinding is not added. The resulting cluster contains just those operations for which 
opbindings exist for all the required routine parameters. For example, as mentioned above, set[any] 
would not have a copy operation because any$copy does not exist and therefore the needed opbinding 
is not present. On the other hand, set[int] does have a copy operation because int$copy does exist. 
Finally, set[any with {copy: foo}], where foo is a procedure that takes an any as an argument and returns 
an any as a result, would have a copy operation.

For an instantiation to be legal it must type check. Type checking is done after the syntactic sugars are 
applied. The types of constant parameters must be included in the declared type, type actuals must be 
types, and the types of the actual routine parameters must be included in the proctypes, lctypes, or 
creatortypes declared in the appropriate where clauses. Of course, the number of parameters declared 
must match the number of actuals passed and with each type actual parameter there must be an 
opbinding for each required routine parameter. If the generator is a cluster, then opbindings must be 
provided for all operations required in the cluster's where clause; opbindings can (but need not) be 
provided for optional parameters. Extra actual routine parameters are illegal.

Because the meaning of an instantiation may depend on the actual routine parameters, type equality 
makes instances with different actual routine parameters distinct types. For example, consider the set 
type generator again; the instance 

set[ array[int] with {equal: array[int]$equal} ]
is not equal to

set[ array[int] with {equal: array[int]$similar} ]

Intuitively these instances should be unequal because the two equal procedures define different 
equivalence classes and therefore the abstract behaviors of the two instances are different. However, 
optional parameters do not effect type equality. For example,

set[ array[int] with {copy: int$copy} ]
and

set[ array[int] with {copy: my_copy} ]

are equal types. This is intuitively justified because in each case set objects behave the same way even 
though different sets are produced when sets are copied in the two cases.

Thus we have the following type equality rule, which defines when two type Specs denote equal types 
(after syntactic sugars are applied). A similar notion is also needed for routine equality. A formal type
identifier is equal only to itself for type checking purposes. Otherwise, two type names denote equal types if they denote the same Description Unit (DU).\textsuperscript{11} Similarly, Argus compares the names of routine formals or the DUs of routines, or checks that they are the same operation in equal types. To decide the equality of two type generator instantiations:

\[
T[t_1 \text{ with } \{ \text{op}_1: \text{act}_1, \ldots, \text{op}_m: \text{act}_m \}, \ldots, t_n \text{ with } \{ \ldots \} ]
\]

and

\[
T'[t_1' \text{ with } \{ \text{op}_1': \text{act}_1', \ldots, \text{op}_m': \text{act}_m' \}, \ldots, t_n' \text{ with } \{ \ldots \} ]
\]

Argus first checks whether:

1. \(T\) and \(T'\) denote the same DU, and whether

2. they have the same number of \textit{type actuals}, and \(t_i\) is equal to \(t_i'\), etc.

Second, any optional parameter \textit{opbindings} in either instantiation are deleted. After this step, Argus checks that for each corresponding \textit{type actual} there is the same number of \textit{opbindings} and that each corresponding \textit{opbinding} is the same. (That is, the corresponding actual routines are equal.) The order of the actual routine parameters does not matter, since Argus matches \textit{opbindings} by operation names. (The definition of routine equality for instantiations of routine generators is similar.) This definition, for example, tells us that

\[
\text{set[ array[Int] with } \{ \text{equal: array[Int]$source} \} ]
\]

is different from

\[
\text{set[ array[\textit{Int}] with } \{ \text{equal: array[\textit{Int}$source} \} ]
\]

(assuming \textit{set} requires an \textit{equal} operation from its type parameter). It also tells us that:

\[
\text{set[ Int with } \{ \text{equal: foo, copy: bar} \} ]
\]

and

\[
\text{set[ Int with } \{ \text{equal: foo, copy: xerox} \} ]
\]

are equal (assuming \textit{copy} is required only by the \textit{set[Int]$source} operation).

This type equality rule allows programmers to control what requirements affect type equality by choosing whether to put them on a cluster or on each operation. A requirement on the cluster should be used whenever the actuals make some difference in the abstraction. For example, in the \textit{set} cluster, the type parameter’s \textit{equal} operation should be required by the cluster as a whole, since using different equality tests for a set’s objects causes the set’s behavior to change.

One can require that a type parameter, say \(t\), be transmissible by stating the requirement:

\[
t \text{ has transmit}
\]

This requirement is regarded as a formal parameter declaration for a special "transmit actual", but Argus does not provide syntax for passing it explicitly. The "transmit actual" is passed implicitly just when the actual type parameter is transmissible and the generator requires it.

\textsuperscript{11}This is name equality unless the type environment has synonyms for types.
12.7. Own Variables

Occasionally it is desirable to have a module that retains information internally between calls. Without such an ability, the information would either have to be reconstructed at every call, which can be expensive (and may even be impossible if the information depends on previous calls), or the information would have to be passed in through arguments, which is undesirable because the information is then subject to uncontrolled modification in other modules (but see also the binding mechanism described in Section 9.8).

Procedures, iterators, handlers, creators, and clusters may all retain information through the use of own variables. An own variable is similar to a normal variable, except that it exists for the life of the program or guardian, rather than being bound to the life of any particular routine activation. Syntactically, own variable declarations must appear immediately after the equates in a routine or cluster body; they cannot appear in bodies nested within statements. Declarations of own variables have the form:

\[
\text{own var ::= own decl}
\]

\[
\begin{align*}
& \text{ | own idn : type spec := expression} \\
& \text{ | own decl , ... := call [ @ primary ]}
\end{align*}
\]

Note that initialization is optional.

The own variables of a module are created when a guardian begins execution or recovers from a crash, and they always start out uninitialized. The own variables of a routine (including cluster operations) are initialized in textual order as part of the first call of an operation of that routine (or the first such call after a crash), before any statements in the body of the routine are executed. Cluster own variables are initialized in textual order, as part of the first call of the first cluster operation to be called (even if the operation does not use the own variables). Cluster own variables are initialized before any operation own variables are initialized. Argus insures that only one process can execute a cluster's or a routine's own variable initializations.

Aside from the placement of their declarations, the time of their initialization, and their lifetime, own variables act just like normal variables and can be used in all the same places. As with normal variables, an attempt to use an uninitialized own variable (if not detected at compile-time) will cause the guardian to crash.

Declarations of own variables in different modules always refer to distinct own variables, and distinct guardians never share own variables. Furthermore, own variable declarations within a parameterized module produce distinct own variables for each distinct instantiation of the module. For a given instantiation of a parameterized cluster, all instantiations of the type's operations share the same set of cluster own variables, but distinct instantiations of parameterized operations have distinct routine own variables.

Declarations of own variables cannot be enclosed by an except statement, so care must be exercised when writing initialization expressions. If an exception is raised by an initialization expression, it will be
treated as an exception raised, but not handled, in the body of the routine whose call caused the initialization to be attempted. Thus, the guardian will crash due to this error.
13. Guardians

This section is concerned with the form and meaning of the modules used to define guardians. Such a
module, called a guardian definition, declares the objects making up the guardian's stable state and
volatile state, and provides implementations for the guardian handlers. It also defines one or more
creators: operations that produce new guardians that behave in accordance with the guardian definition.
In addition, a guardian definition may provide functions that may be executed independently of the
recovery code to restore the volatile state when the guardian is restarted after a crash.

The syntactic form of a guardian definition is as follows:

\[
\begin{align*}
\text{guardian} \left( \text{params} \right) \text{ is } \text{idn} \ldots \left[ \text{handles} \text{idn} \ldots \right] \left\{ \text{where} \right\} \text{idn} \\
\{ \text{equals} \} \text{idn} \\
\{ \text{state_decl} \} \\
\{ \text{restor_body} \text{ and} \} \\
\{ \text{backgrnd_body} \text{ and} \} \\
\{ \text{operation} \} \text{ creator} \{ \text{operation} \} \\
\text{end idn}
\end{align*}
\]

where

\[
\text{operation} \rightarrow \text{creator} \\
\| \text{handler} \\
\| \text{routine}
\]

The initial idn names the guardian type or type generator (as explained in Section 9.4) and must agree
with the final idn. The guardian handler contains two idn lists. The first, following the, gives the names of
the creators, which can be called to create and initialize new guardians (the objects belonging to the
guardian type). The second, following the, gives the names of the handlers that can be called
on those guardian objects. The names of all operations must be unique. Creators may not be named
equal, similar, copy, or gen whereas \text{is} is the name of a handler. See Section 9.4 for the initial type
defined by a guardian definition. See Section 12.5 for the meaning of guardians having \text{params} and \text{where}
clauses.

The remaining portions of the guardian definition are discussed in the subsections below.

13.1. The Guardian State

The state_decls of the guardian definition declare a number of variables (with optional initialization):

\[
\text{state_decl} \rightarrow \text{idn} \left( \text{stable} \right) \text{ decl} \\
\| \left[ \text{stable} \right] \text{idn} : \text{type spec} \rightarrow \text{expression} \\
\| \left[ \text{stable} \right] \text{decl}, \ldots \rightarrow \text{call}
\]

The scope of these declarations is the entire guardian definition. The objects reachable from variables
declared to be \text{stable} survive crashes of the guardian, while other objects do not.

For example, if the state_decls were:
stable buffer: atomic_array[int] := atomic_array[int] new()
cache: array[int] := array[int] new()

then the atomic_array object denoted by buffer would survive a guardian crash, but the array object denoted by cache would not. See Section 13.3 for more details of crash recovery. Volatile variables can be assigned wherever an assignment statement is legal. However, stable variables may only be assigned by an initialization when declared or in the body of a creator. The initializations of both stable and volatile variables are executed within an action, as described below. However, the stable variables are not reinitialized upon crash recovery, whereas volatile variables are reinitialized upon crash recovery.

Stable variables should denote resilient objects (see Section 15.2), because only resilient data objects (reachable from the stable variables) are written to stable storage when a topaction commits. (This can be ensured by having stable variables only denote objects of an atomic type or objects protected by mutex.) Non-resilient objects stored in stable variables are only written to stable storage once, when the guardian is created. Furthermore, the stable variables should usually denote atomic objects, because the stable variables are potentially shared by all the actions in a guardian.

13.2. Creators

A guardian definition must provide one or more creators. The names of these creators must be listed in the guardian header (internal creators are not allowed); each such name must correspond to a single creator definition appearing in the body of the guardian definition.

A creator definition has the same form as a procedure definition, except that creators cannot be parameterized, and the reserved word creator is used in place of proc:

```
iden = creator ([ args ] [ returns ] [ signals ]
    routine_body
   end iden
```

The initial iden names the creator and must agree with the final iden. The types of all arguments and all results (normal and exceptional) must be transmissible.

A creator is an object of some creator type. This type is derived from the creator heading by removing the creator name, rewriting the formal argument declarations with one iden per decl, deleting the inames of all formal arguments, deleting any failure or unavailable signals, and finally, replacing creator by creatortype. The signals failure(string) and unavailable(string) are implicit in every creator type (since they can arise from any creator call). However, if these signals are raised explicitly by a creator, they must be listed in the signals clause with string result types.

The semantics of a creator call are explained in Section 8.4. Typically, the body of a creator will initialize some stable and volatile variables. It can also return the name of the guardian being created using the expression self. Since the creator (and the state initialization) runs as an action, the creator terminates by committing or aborting. If it aborts, the guardian is destroyed. If it commits, the guardian begins to accept handler calls, and runs the background code, if any (see below). If an ancestor of the creator aborts, the guardian is destroyed. If the creator and all its ancestors commit, the guardian becomes permanent, and will survive subsequent crashes.
13.3. Crash Recovery

Once a guardian becomes permanent, it will be recreated automatically after a crash with its stable variables initialized to the same state they were in at the last topaction commit before the crash. The volatile variables are then initialized (in declaration order) by a topaction. To aid in this reinitialization, the guardian definition can provide a `recover section`:

```
recover body end
```

to be run, as part of this topaction, after the initializations attached to the volatile variable declarations are performed. The recover section commits when control reaches the end of the body, or when a return statement is executed. The recover section may abort by executing an `abort return` statement or as a result of an unhandled exception. The guardian crashes if the recover section aborts.

13.4. Background Tasks

Tasks that must be performed periodically, independent of handler calls, can be defined by a `background section`:

```
background body end
```

The system creates a process to run this body as soon as creation or recovery commits successfully. The body of the background section does not run as an action; typically it will perform a sequence of topactions.

If the background process finishes executing its body (either by reaching the end of the block or by returning), the process terminates, but the guardian continues to execute incoming handler calls.

13.5. Handlers and Other Routines

Typically, the principal purpose of a guardian is to execute incoming handler calls. A guardian accepts handler calls as soon as creation or recovery commits.

The guardian header lists the names of the externally available handlers. Each handler listed must be defined by a handler definition. Additional handler definitions may also be given, but these handlers can be named only within the guardian to which they belong.

A handler definition has the same form as a procedure definition, except that handlers cannot be parameterized, and the reserved word `handler` is used in place of `proc`:

```
hidn = handler ([args]) [returns] [signals]
  routine_body
end hidn
```

The initial `hidn` names the handler and must agree with the final `hidn`. The types of all arguments and all results (normal and exceptional) must be transmissible.

A handler is an object of some handler type. This type is derived from the handler heading by removing the handler name, rewriting the formal argument declarations with one `hidn` per decl, deleting the
ideas of all formal arguments, deleting any failures or unwanted values, and finally, replacing handler by handler_type. The signals (functions) and resources generated are unique to every handler type. However, if these signals are reused explicitly by a handler, they must return to the signal classes, with strings as their result type.

As explained in Section 8.3, a handler cell runs as a subroutine of its caller, and arguments and results are passed by value. A new process is created at the boundary of the handler activation. Since the handler activation is an active, it terminates by案終される処理に変更

A guardian definition may also contain procedures and handler definitions. These procedures and handlers may be called only within the guardian is within any enclosing.

13.6. Guardian Lifetime and Destruction

A guardian does not become permanent until the moment when the greater activation that initialized the guardian) comes to the top. If the greater (or any superior greater) statement is true, the guardian will be destroyed.

Once a guardian becomes permanent, it will survive all activations (within or outside) and thus may live forever. However, a change in scope may be removed by a greater statement that destroys the guardian and never destroy another, but a greater statement that destroys the guardian will be destroyed. (See Section 6.10.) (A statement greater than a greater statement.)

A short-lived guardian can be implemented by using background code on the form:

```
background terminate;  // The background code starts to run as soon as the greater enters. The guardian is deleted, however, until the greater code comes to the top, so the greater must check to see if the greater of the new guardian before it is destroyed. (If an ancestor of the greater code, the guardian will be destroyed automatically.)
```

The following is an example of a handler for destroying a permanent guardian:

```
finish = handler (...) returns (...) signals (not_authorized)
...
  terminate
  return(...)
  and finish
```

Here, finish might check whether its caller is authorized to make this request, and signal not_authorized if not. Otherwise it returns the vital state information to its superior to form its information.

13.7. An Example

To illustrate how much of the components of a guardian definition are used, an example of a simple guardian is given in Figure 13-1. An action can use a greater statement or superior until after the action has come to the top. The greater then passes the control to other guardians for
consumption. The spooler provides an operation for adding (object, consumer) pairs, and for destroying the guardian.

Figure 13-1: Spooler Guardian

```plaintext
spooler = guardian [t: type] la create handles enq, finish
    where t has transmit

utype = handler type (t)
entry = struct[object: t, consumer: utype]
queue = semiquene[entry]

stable state: queue := queue$create()

background
    while true do
        enter topaction
            e: entry := queue$deq(state)
            e.consumer(e.object)
            except when unavailable ("": abort leave end
            end except when failure, unavailable ("": end
        end
    end

create = creator () returns (spooler[t])
    return(self)
    end create

enq = handler (item: t, user: utype)
    queue$enq(state, entry$[object: item, consumer: user])
    end enq

finish = handler ()
    terminate
    end finish

end spooler
```

The spooler guardian is parameterized by the type of object to be stored. The enq handler takes an object of this type, and a handler for sending the object to the consumer, and adds this information to the stable state of the spooler. This state is an object of the semiquene abstract data type\(^{12}\). Each entry in the semiquene is a structure containing a stored object and its corresponding consumer handler. The background code of the guardian runs an infinite loop that starts a topaction, removes an entry from the queue, and sends the object using the associated handler.

Note that an unavailable exception arising from this handler call is caught inside the topaction, so that an explicit abort can be performed. If the exception were caught outside the topaction, it would cause the

---

topaction to commit, and the entry would be removed without being consumed. Note also that failure is caught outside the topaction, since if an encode were to fail, or if the guardian did not exist, the background process might aimlessly loop forever, because it would not be able to remove that entry.

A more extended example of a distributed system appears in the paper Liskov, B. and Scheffler, R., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," ACM Transactions on Programming Languages and Systems, volume 5, number 3, (July 1983), pages 381-404.
14. Transmissibility

A type is said to be transmissible if it defines a transmit operation that allows the values of its objects to be sent in messages or stored in image objects. Only objects of transmissible type may be used as arguments to header calls or create calls. This section describes how transmission is defined for the Argus built-in types and for user-defined types.

14.1. The Transmit Operation

Transmissibility is a property of a data abstraction and must be stated in the specification of that abstraction. A transmissible data type \( T \) can be characterized as having an additional operation,

\[
\text{transmit} = \text{preotype}(T) \times \text{subtype}(T) \times \text{transmit}(T)
\]

which is called implicitly during message transmission. When an object, transmit produces a different object, which may even reside at a different location from the original. The relation between the original object and the transmitted object is defined by the operation \( \text{transmit} \). Although the exact specification of transmit is type dependent, the values of the two objects will typically be equal. (Value equality is also part of a type's specification, see the discussion in section 12.2 of the CLU Reference Manual<sup>16</sup>). The transmit operation for a type runs exactly as specified for the type for its objects.

14.2. Transmission for Built-in Types

The unstructured built-in types (int, char, float, ...) are transmissible, with the exception of preotype, subtype, and any. The transmit operation of the transmit operations preserves value equality, with the exception of the real type, which, because of possible negative elements, guarantees only that the two values differ by very little.

The structured types (instances of array, struct, atomic, record, ...) are transmissible if and only if all their type parameters are transmissible. The transmit operation for a structured type is defined in terms of the transmit operations of the component types. For example, if an object \( x \) is an array containing elements of type \( T \), then the transmit operation for \( x \) creates a new array \( y \) with the same bounds as the original, and with elements:

\[
y[i] = \text{transmit}(x[i])
\]

Thus transmission of the built-in structured types will preserve value equality only if transmission of the component types does.

The transmit operation for matrix\( [T] \) acquires and holds the lock during the transmission (actually, during the encoding, see below) of the contained object.

---

14.3. Transmit for Abstract Types

The type implemented by a cluster is transmissible if the reserved word transmit appears in the la-list at the head of the cluster. Unlike the other operations provided by a type, the transmit operation cannot be called directly by users, and in fact is not implemented directly in the cluster. Instead, transmit is implemented indirectly in the following way. Each transmissible type is given a canonical representation, called its external representation type. The external representation type of an abstract type T is any convenient transmissible type XT. This type can be another abstract type if desired; there is no requirement that XT be a built-in type. Intuitively, the meaning of the external representation is that values of type XT will be used in messages to represent values of type T. The choice of external representation type is made for the abstract type as a whole and must be used in every implementation of that type. (There are currently no provisions for changing the external representation of a type once it has been established in the library.)

Each implementation of the abstract type T must provide two operations to map between values of the abstract type and values of the external representation type. There is an operation

\[ \text{encode} = \text{proc (a: T) returns (XT) [ signals (failure(string)) ]} \]

to map from T values to XT values (for sending messages) and an operation

\[ \text{decode} = \text{proc (x: XT) returns (T) [ signals (failure(string)) ]} \]

to map from XT values to T values (for receiving messages). The transmit operation for T is defined by the following identity:

\[ T\text{transmit} (x) = T\text{decode} (XT\text{transmit} (T\text{encode}(x))) \]

Intuitively, the correctness requirement for encode and decode is that they preserve the abstract T values: encode maps a value of type T into the XT value that represents it, while decode performs the reverse mapping\(^{14}\).

Encode and decode are called implicitly by the Argus system during handler and creator calls. If encode and decode do not appear in the cluster's la-list, then they will be accessible to the Argus system, but may not be named directly by users of the type. A failure exception raised by one of these operations will be caught by the Argus system and resigalled to the caller (see Section 8.3).

An abstract type's encode and decode operations should not cause any side effects. This is because the number of calls to encode or decode is unpredictable, since arguments or results may be encoded and decoded several times as the system tries to establish communication. In addition, verifying the correctness of transmission is easier if encode and decode are simply transformations to and from the external representation.

When defining a parameterized module (see Section 12.5), it may be necessary to require a type parameter to be transmissible. A special type restriction:

has transmit

is provided for this purpose. To permit instantiation only with transmissible type parameters, this
restriction should appear in the where clause of the cluster. Alternatively, by placing identical where
clauses in the headings of encode and decode procedures, one can ensure that an instantiation of the
cluster is transmissible only if the type parameters are transmissible (see Section 12.5).

As an example, Figure 14-1 shows part of a cluster defining a key-item table that stores pairs of values,
where one value (the key) is used to retrieve the other (the item). The key-item table type has operations
for creating empty tables, inserting pairs, retrieving the item paired with a given key, deleting pairs, and
iterating through all key-item pairs. The table is represented by a sorted binary tree, and its external
representation is an array of key-item pairs. The table type is transmissible only if both type parameters
are transmissible.

Figure 14-1: Partial implementation of table.

table = cluster [key, item: type] is create, insert, lookup, allpairs, delete, transmit, ...
    where key has lt; proctype (key, key) returns (bool),
              equal: proctype (key, key) returns (bool)

    pair = record [k: key, i: item]
    nod = record [k: key, i: item, left, right: table[key, item]]
    rep = variant [empty: null, some: nod]
    xrep = array [pair] % the external representation type

    % The internal representation is a sorted binary tree. All pairs in the table
    % to the left (right) of a node have keys less than (greater than) the key in
    % that node.

    % ... other operations omitted

    encode = proc (t: table[key, item]) returns (xrep)
        where key has transmit, item has transmit
        xr: xrep := xrep$new() % create an empty array
        % use allpairs to extract the pairs from the tree
        for p: pair in allpairs(t) do
            % Add the pair to the high end of the array.
            xrep$saddh(xr, p)
        end
    return(xr)
end encode

    decode = proc (xtbl: xrep) returns (table[key, item])
        where key has transmit, item has transmit
        t: table[key, item] := create() % create empty table
        for p: pair in xrep$elements(xr) do
            % xrep$elements yields all elements of array xr
            insert(t, p.key, p.item) % enter pair in table
        end
    return(t)
end decode

end table
14.4. Sharing

When an object of structured built-in type is encoded and decoded, sharing among the object's components is preserved. For example, let \( a \) be an array[7] object such that \( a[7] \) and \( a[1] \) refer to a single object of type \( T \). If \( a2 \) is an array[7] object created by transmitting \( a \), then \( a2[7] \) and \( a2[1] \) also name a single object of type \( T \).

All sharing is preserved among all components of multiple objects of built-in type when those objects are encoded together. Thus, sharing is preserved for objects that are arguments of the same remote call or are results of the same remote call, unless the arguments are encoded at different times (see the discussion of the blind expression in Section 9.8). For example, let \( a \) and \( b \) be array[7] objects such that \( a[7] \) and \( b[7] \) refer to a single object of type \( T \). If \( a2 \) and \( b2 \) are arrays created by sending \( a \) and \( b \) as arguments in a single handler call, then \( a2[7] \) and \( b2[7] \) also refer to a single object.

Whether an abstract type’s transmit operation preserves sharing is part of that type’s specification, but sharing should usually be preserved for abstract types. In the key-item table implementation of Figure 14-1, there are two types of sharing that should be preserved: sharing of keys and items among multiple tables sent in a single message, and sharing of items bound to the same key in a single table. The key-item table example shows how to implement an abstract type whose transmission preserves sharing by choosing an external representation type whose transmit operation preserves sharing.

Care must be taken when the references among objects to be transmitted are cyclic, as in a circular list. Decoding such objects can result in a failure exception unless encode and decode are implemented in one of two ways:

1. the internal and external representation types are identical and encode and decode return their argument object without modifying it or accessing its components, or

2. the external representation object must be free of cycles.
15. Atomic Types

In Argus, atomicity is enforced by the objects shared among actions, rather than by the individual actions themselves. Types whose objects ensure atomicity of the actions sharing them are called atomic types; objects of atomic types are called atomic objects. In this chapter we define what it means for a type to be atomic and describe the mechanisms provided by Argus to support the implementation of atomic types.

Atomicity consists of two properties: serializability and recoverability. An atomic type’s objects must synchronize actions to ensure that the actions are serializable. An atomic type’s objects must also recover from actions that abort to ensure that actions appear to execute either completely or not at all.

In addition, an atomic type must be resilient: the type must be implemented so that its objects can be saved on stable storage. This ensures that the effects of an action that commits to the top (that is, an action that commits, as do all of its ancestors) will survive crashes.

This chapter provides definitions of the mechanisms used for user-defined types in Argus. For example implementations, see Weihl, W. and Liskov, B., "Implementation of Resilient, Atomic Data Types," ACM Transactions on Programming Languages and Systems, volume 7, number 2 (April 1985), pages 244-269.

The remainder of this chapter is organized as follows. In Section 15.1 and Section 15.2, we present the details of the mechanisms. Section 15.1 focuses on synchronization and recovery of actions, while Section 15.2 deals primarily with resilience. In Section 15.3, we discuss some guidelines to keep in mind when using the mechanisms described in Section 15.1 and Section 15.2. In Sections 15.4 and 15.5, we define more precisely what it means for a type to be atomic. Finally, in 15.6, we discuss some details that are important for user-defined atomic types that are implemented using multiple mutexes.

15.1. Action Synchronization and Recovery

In this section we describe the mechanisms provided by Argus to support synchronization and recovery of actions. These mechanisms are designed specifically to support implementations of atomic types that allow highly concurrent access to objects.

Like a non-atomic type, an atomic type is implemented by a cluster that defines a representation for the objects of the type, and an implementation for each operation of the type in terms of that representation. However, the implementation of an atomic type must solve some problems that do not occur for ordinary types, namely: synchronizing concurrent actions, making visible to other actions the effects of committed actions, hiding the effects of aborted actions, and providing resilience against crashes.

An implementation of a user-defined atomic type must be able to find out about the commits and aborts of actions. In Argus, implementations use objects of built-in atomic types for this purpose. The representation of a user-defined atomic type is typically a combination of atomic and non-atomic objects;
the non-atomic objects are used to hold information that can be accessed by concurrent actions, while the atomic objects contain information that allows the non-atomic objects to be monitored properly. The fact that atomic objects can be used to answer the following question: can we ensure that caused a particular change to the representation:

- commit (so the new information is now available to other actions),
- abort (so the change should be forgotten), or
- is it still active (so the information cannot be released yet)?

The operations available on atomic objects have been extended to support this type of use; in particular, the can_read and can_write operations are now defined as optional versions, and the restart and tegnament statements are intended to be used for this purpose. (We do not expect user-defined atomic types to support such operations, however.)

The use of atomic objects in the representation permits operation implementations to discover what happened to previous actions and to synchronize concurrently. Moreover, since part of the representation of a user-defined atomic object may be an atomic object, this also serves as a way to synchronize concurrent operation implementations on user-defined operations.

Synchronization for non-atomic data is provided by the communication barrier. As discussed in Section 6.7, a mutex(?) object is essentially a container for an object that must not be used to provide mutual exclusion for the contained object. The function mutex(make_mutex) in Section 19.19, is used to gain possession of a mutex object. The function mutex_release in Section 19.19 need not have access to a mutex object while it executes the body of the mutex. The function mutex_unlock unlocks an object that it needs to wait for (e.g., a semaphore). The function mutex_lock acquires the object. The pause statement, described in Section 19.17, can be used to block execution of an operation until it can acquire. The pause statement, described in Section 19.17, can be used to block execution of an operation until it can acquire. The pause statement, described in Section 19.17, can be used to block execution of an operation until it can acquire.

15.2. Ressources

If a user-defined atomic object is accessible from the state machine at some question, it should be written to stable storage whenever an atomic object that is accessible from the state machine is written to stable storage. In this section, we discuss how user-defined atomic objects are implemented in the system. We describe how the objects are written to stable storage properly. Each set of implementations for each user-defined atomic object is implemented separately.

In addition to its use for synchronizing user processes, mutex is used for these other functions:
- notifying the system when information needs to be synchronized over the network when information is written to stable storage, and ensuring that the system does not overwrite newer information.

To minimize the amount of information that must be written to stable storage, when an editing session commences, the Argus system only copies new information to stable storage at the start of the session. It copies accessible objects modified or made empty; accessible objects that are not modified are not copied to the top. For mutex objects, it also copies every accessible object to stable storage. In addition, the mutex operation
changed = proc (m: mutex{T})
is provided for notifying the system that an existing mutex object should be written to stable storage. Calling this operation will cause the object to be written to stable storage (assuming it is accessible) by the time the action that executed the changed operation commits to the top. Sometime after the action calls changed, and before its top-level ancestor commits, the system will copy the mutex object to stable storage. Changed must be called from a process running an action.

Mutex objects also define how much information must be written to stable storage. Copying a mutex object involves copying the contained object. By choosing the proper granularity of mutex objects the user can control how much data must be written to stable storage at a time. For example, a large database can be broken into partitions that are written to stable storage independently by dividing it among several mutex objects. Such a division can be used to limit the amount of data written to stable storage by calling changed only for those partitions actually modified by a committing action.

In copying a mutex object, the system will copy all objects reachable from it, excluding other mutex or built-in atomic objects. A contained mutex or built-in atomic object will be copied only if necessary; that is, only if it is:

- a mutex object for which (a descendant of) the completing action called the changed operation,
- a built-in atomic object that was modified by the action, or
- a newly accessible object for which no stable copy exists.

Furthermore, the component is copied independently of the containing mutex object; they may be copied in either order (or simultaneously), subject to the constraint that the system cannot copy a mutex object without first gaining possession of it.

Finally, mutex objects can be used to ensure that information is in a consistent state when it is written to stable storage. The system will gain possession of a mutex object before writing it to stable storage. By making all modifications to mutex objects inside seize statements, the user’s code can prevent the system from copying a mutex object when it is in an inconsistent state.

Some details of the effect of changed are important for atomic types that are implemented as multiple mutexes. These details are presented in Section 15.6.

15.3. Guidelines
This section discusses some guidelines to be followed when implementing atomic types. There are additional guidelines to follow when multiple mutexes are used to implement an atomic type; those guidelines are discussed in Section 15.6.

An important concept for describing the resilience of user-defined atomic types is synchrony. An object is synchronous if it is not possible to observe that any portion of the object is copied to stable storage at a different time from any other portion. For example, an object of type array[mutex[int]] would not be
synchronous, because elements of the array can be copied at different times. A type is synchronous if all of its objects are synchronous. Whether a type is synchronous or not is an important property of its behavior and should be stated in its specification. The built-in atomic types are synchronous; user-defined types must also be synchronous if they are to be atomic.

To ensure the resilience and serializability of a user-defined atomic type independently of how it is used, the form of the rep for an atomic type should be one of the following possibilities.

1. The rep is itself atomic. Note that mutex is not an atomic type.

2. The rep is mutex[t] where t is a synchronous type. For example, t could be atomic, or it could be the representation of an atomic type, if the operations on the this fictitious atomic type are coded in-line so that the entire type behaves atomically.

3. The rep is an atomic collection of mutex types containing synchronous types.

4. The rep is a mutable collection of synchronous types, and objects of the representation type are never modified after they are initialized. That is, mutation may be used to create the initial state of such an object, but once this has been done the object must never be modified.

When using mutex objects, there are a few rules to remember. First, changed must be called after the last modification (on behalf of some action) to the contained object. This is true because the Argus system is free to copy the mutex to stable storage as soon as changed has been called.

In addition, changed should be called even if the object is not accessible from the stable variables of a guardian. In part this rule is just an example of separation of concerns: the implementation of the atomic type should be done independently of any assumptions about how the object will be used. Therefore the type should be implemented as if its objects were accessible from the stable variables of some guardian. However, in addition, if this rule is not followed, it is possible that stable storage will not be updated properly. This situation can occur if an object was accessible, then becomes inaccessible, and later becomes accessible again. The system guarantees that no problems arise if changed is always called after the last modification to the object.

Mutex objects should not share data with one another, unless the shared data is atomic or mutex. One reason for this rule is that in copying mutex objects to stable storage Argus does not preserve this kind of sharing.

A final point about mutex objects is that it is unwise to do any activity that is likely to take a long time inside a seize statement. For example, a handler call should not be done from inside a seize statement if possible. Also, it is unwise to wait for a lock inside a seize unless the programmer can be certain that the lock is available or will be soon. Otherwise, a deadlock may occur. An example of where waiting for a lock in a nested seize statement is safe is where all processes seize the two mutex objects in the same order.
15.4. A Prescription for Atomicity

In this section, we discuss how to decide how much concurrency is possible in implementing an atomic type. In writing specifications for atomic types, we have found it helpful to pin down the behavior of the operations, initially assuming no concurrency and no failures, and to deal with concurrency and failures later. In other words, we imagine that the objects will exist in an environment in which all actions are executed sequentially, and in which actions never abort.

Although a sequential specification of this sort does not say anything explicit about permissible concurrency, it does impose limits on how much concurrency can be provided. Implementations can differ in how much concurrency is provided, but no implementation can exceed these limits. Therefore, it is important to understand what the limits are.

This section and the following section together provide a precise definition of permissible concurrency for an atomic type. This definition is based on two facts about Argus and the way it supports implementations of atomic type. First, in implementing an atomic type, it is only necessary to be concerned about active actions. Once an action has committed to the top, it is not possible for it to be aborted later, and its changes to atomic objects become visible to other actions. So, for example, an implementation of an atomic type needs to prevent one action from observing the modifications of other actions that are still active, but it does not have to prevent an action from observing modifications by actions that have already committed. Second, the only method available to an atomic type for controlling the activities of actions is to delay actions while they are executing operations of the type. An atomic type cannot prevent an action from calling an operation, although it can prevent that call from proceeding. Also, an atomic type cannot prevent an action that previously finished a call of an operation from completing either by committing or by aborting.

Given the sequential specification of the operations of a type, these facts lead to two constraints on the concurrency permitted among actions using the type. While an implementation can allow no more concurrency than permitted by these constraints, some implementations, like that for the built-in type generator atomic__array (see Section II.10), may allow less concurrency than permitted by their sequential specifications and our concurrency constraints.

The first constraint is that
- an action can observe the effects of other actions only if those actions committed relative to the first action.

This constraint implies that the results returned by operations executed by one action can reflect changes made by operations executed by other actions only if those actions committed relative to the first action. For example, in an atomic array a, if one action performs a store(a, 3, 7), a second (unrelated) action can receive the answer "7" from a call of fetch(a, 3) only if the first action committed to the top. If the first action is still active, the second action must be delayed until the first action completes. This first constraint supports recoverability since it ensures that effects of aborted actions cannot be observed by other actions. It also supports serializability, since it prevents concurrent actions from observing one another's changes.
However, more is needed for serializability. Thus, we have our second constraint:

- operations executed by one action cannot invalidate the results of operations executed by a concurrent action.

For example, suppose an action $A$ executes the `size` operation on an atomic array object, receiving $n$ as the result. Now suppose another action $B$ is permitted to execute `addh`. The `addh` operation will increase the size of the array to $n+1$, invalidating the results of the `size` operation executed by $A$. Since $A$ observed the state of the array before $B$ executed `addh`, $A$ must precede $B$ in any sequential execution of the actions (since sequential executions must be consistent with the sequential specifications of the objects). Now suppose that $B$ commits. By assumption, $A$ cannot be prevented from seeing the effects of $B$. If $A$ observes any effect of $B$, it will have to follow $B$ in any sequential execution. Since $A$ cannot both precede and follow $B$ in a sequential execution, serializability would be violated. Thus, once $A$ executes `size`, an action that calls `addh` must be delayed until $A$ completes.

## 15.5. Commuting Operations

To state our requirements more precisely, consider a simple situation involving two concurrent actions each executing a single operation on a shared atomic object $X$. (The actions may be executing operations on other shared objects also, but in Argus each object must individually ensure the atomicity of the actions using it, so we focus on the operations involving a single object.) A fairly simple condition that guarantees serializability is the following. Suppose $X$ is an object of type $T$. $X$ has a current state determined by the operations performed by previously committed actions. Suppose $O_1$ and $O_2$ are two executions of operations on $X$ in its current state. ($O_1$ and $O_2$ might be executions of the same operation or different operations.) If $O_1$ has been executed by an action $A$ and $A$ has not yet committed or aborted, $O_2$ can be performed by a concurrent action $B$ only if $O_1$ and $O_2$ commute: given the current state of $X$, the effect (as described by the sequential specification of $T$) of performing $O_1$ on $X$ followed by $O_2$ is the same as performing $O_2$ on $X$ followed by $O_1$. It is important to realize that when we say "effect" we include both the results returned and any modifications to the state of $X$.

The intuitive explanation of why the above condition works is as follows. Suppose $O_1$ and $O_2$ are performed by concurrent actions $A$ and $B$ at $X$. If $O_1$ and $O_2$ commute, then the order in which $A$ and $B$ are serialized globally does not matter at $X$. If $A$ is serialized before $B$, then the local effect at $X$ is as if $O_1$ were performed before $O_2$, while if $B$ is serialized before $A$, the local effect is as if $O_2$ were performed before $O_1$. But these two effects are the same since $O_1$ and $O_2$ commute.

The common method of dividing operations into readers and writers and using read/write locking works because it allows operations to be executed by concurrent actions only when the operations commute. More concurrency is possible with our commutativity condition than with readers/writers because the meaning of the individual operations and the arguments of the calls can be considered. For example, calls of the atomic array operation `addh` always commute with calls of `addl`, yet both these operations are writers. As another example, `store(X, i, e_1)` and `store(X, j, e_2)` commute if $i \neq j$.

We require only that $O_1$ and $O_2$ commute when they are executed starting in the current state.
15.5 Commuting Operations

Consider a bank account object, with operations to deposit a sum of money, to withdraw a sum of money (with the possible result that it signals insufficient funds if the current balance is less than the sum requested), and to examine the current balance. Two withdraw operations, say for amounts \( m \) and \( n \), do not commute when the current balance is the maximum of \( m \) and \( n \): either operation when executed in this state will succeed in withdrawing the requested sum, but the other operation must signal insufficient funds if executed in the resulting state. They do commute whenever the current balance is at least the sum of \( m \) and \( n \). Thus if one action has executed a withdraw operation, our condition allows a second action to execute another withdraw operation while the first action is still active as long as there are sufficient funds to satisfy both withdrawal requests.

Our condition must be extended to cover two additional cases. First, there may be more than two concurrent actions at a time. Suppose \( A_1,\ldots,A_n \) are concurrent actions, each performing a single operation execution \( O_1,\ldots,O_n \) respectively, on \( X \). (As before, the concurrent actions may be sharing other objects as well.) Since \( A_1,\ldots,A_n \) are permitted to be concurrent at \( X \), there is no local control over the order in which they may appear to occur. Therefore, all possible orders must have the same effect at \( X \). This is true provided that all permutations of \( O_1,\ldots,O_n \) have the same effect when executed in the current state, where effect includes both results obtained and modifications to \( X \).

The second extension acknowledges that actions can perform sequences of operation executions. Consider concurrent actions \( A_1,\ldots,A_n \) each performing a sequence \( S_1,\ldots,S_n \) respectively, of operation executions. This is permissible if all sequences \( S_1,\ldots,S_n \) obtained by concatenating the sequences \( S_1,\ldots,S_n \) in some order, produce the same effect. For example, suppose action \( A \) executed \( addh \) followed by \( remh \) on an array. This sequence of operations has no net effect on the array. It is then permissible to allow a concurrent action \( B \) to execute \( size \) on the same array, provided the answer returned is the size of the array before \( A \) executed \( addh \) or after it executed \( remh \).

Note that in requiring certain sequences of operations to have the same effect, we are considering the effect of the operations as described by the specification of the type. Thus we are concerned with the abstract state of \( X \), and not with the concrete state of its storage representation. Therefore, we may allow two operations (or sequences of operations) that do commute in terms of their effect on the abstract state of \( X \) to be performed by concurrent actions, even though they do not commute in terms of their effect on the representation of \( X \). This distinction between an abstraction and its implementation is crucial in achieving reasonable performance.

It is important to realize that the constraints that are imposed by atomicity based on the sequential specification of a type are only an upper bound on the concurrency that an implementation may provide. A specification may contain additional constraints that further constrain implementations; these constraints may be essential for showing that actions using the type do not deadlock, or for showing other kinds of termination properties. For example, the specification of the built-in atomic types explicitly describes the locking rules used by their implementations; users of these types are guaranteed that the built-in atomic types will not permit more concurrency than allowed by these rules (for instance, actions writing different components of an array, or different fields of a record, cannot do so concurrently).
15.6. Multiple Mutexes

Section 15.5 presented a discussion of copying mutex objects to stable storage. That discussion is adequate for simple implementations that use just one mutex object. Sometimes, however, it is desirable to use more than one mutex object in representing a contained-in relationship; for example, a partitioned database would be implemented this way. Some important points require an understanding of some details that could be ignored when just one mutex object was used in the representation. In particular, the implementor must understand the effects of concurrent updates of mutex objects and some problems that can arise because copying to stable storage is uncertain.

The writing of mutex objects to stable storage is necessary for each operation at each location: either all mutexes modified by an action at a location are written to stable storage, or none of them are. That is, if an action modified more than one mutex object at a location, then after a crash either all of these objects will be recovered, or none of them will be. This makes it easier to preserve consistency among multiple mutex objects. However, even if all mutex objects, the new versions of mutexes if modified may still be recovered from stable storage, and the system may only guarantee that if any new versions are recovered, all of them will be. The point is that when two actions are taken at a location, it is possible that new versions will be installed at some of the locations but not others.

Although mutex objects modified by a single action are handled as stable storage as a group, the copies are made one at a time. Incremental copying also works with multiple actions. The true state of an object usually includes the state of the contained objects as well, and the representation invariant expressing a consistency condition on an action can be used to ensure the states of contained objects. For example, suppose we had an action that involved actions that modified contained objects to enqueue and dequeue different items (that is, the actions commute and they operations commute so long as they involve different objects in the containment). Then consider an implementation of a double-queue that (for some reason) kept two copies of these contents and was represented by:

rep = struct [first, second: contents]

where the representation invariant required that the states of the two containers be the same. Now suppose the system is handling the top-level actions of some mutex. If two modified both containers contained in the double-queue, and while this is occurring a second action is modifying these containers. Then it is possible that when the first operation is written to stable storage it contains S's changes, but when the second operation is written to stable storage it does not contain S's changes. Therefore, the information in stable storage appears not to satisfy the representation invariant of the double-queue.

However, the representation invariant of the double-queue really is satisfied, for the following reason. First note that the implementation in stable storage is only of interest after a crash. So suppose there is a crash. Now there are two possibilities:

\[\text{rep = struct [first, second: contents]}\]

1. Before that crash, B also committed to the top. In this case the data read back from stable storage is, in fact, consistent, since it must reflect B's changes to both the first and second semiquotes.

2. B aborted or had not yet committed before the crash. In either case, B aborts. Therefore, the changes made to the first semiquete by B will be hidden by the semiquete implementation: at the abstract level, the two semiquotes do have the same state.

The point of the above example is that if the objects being written to stable storage are atomic, then the fact that they are written incrementally causes no problems.

On the other hand, when an atomic type is implemented with a representation consisting of several mutex objects, the programmer must be aware that these objects are written to stable storage incrementally, and care must be taken to ensure that the representation invariant is still preserved and that information is not lost in spite of incremental writing. If the implementation of a type requires that one mutex object (call it M1) be written to stable storage before another (call it M2), then the write of M1 must be contained in an action that commits to the top before the action that writes M2 is run.
Appendix I
Syntax

We use an extended BNF grammar to define the syntax. The general form of a production is

nonterminal ::= alternative
              | alternative
              | ...
              | alternative

The following extensions are used:

a, ... a list of one or more as separated by commas: "a", "a, a" or "a, a, a", etc.
{ a } a sequence of zero or more as: " * " or "a * " or "a a", etc.
[ a ] an optional a: " * " or " a ".

Nonterminal symbols appear in normal face. Reserved words appear in bold face. All other terminal
symbols are nonalphabetic and appear in normal face.

module ::= { equate } equates
         | { equate } guardian
         | { equate } procedure
         | { equate } memory
         | { equate } cluster

equates ::= idn = equates [ params [ where ] ]
             equate { equate }
             end idn

guardian ::= idn = guardian [ params ] to idn, ... [ handlers idn, ... ] [ where ]
             { equate }
             { state_def }
             { resource body end }
             { background body end }
             { operation } creator { operation }
             end idn

cluster ::= idn = cluster [ params ] to equity, ... [ where ]
           { equate } exp = open equity { equate }
           { close_var }
            routine { routine }
           end idn
operation ::= creator
  | handler
  | routine

routine ::= procedure
  | iterator

procedure ::= idn = proc [ parms ] args [ returns ] [ signals ] [ where ]
  routine_body
  end idn

iterator ::= idn = iter [ parms ] args [ yields ] [ signals ] [ where ]
  routine_body
  end idn

creator ::= idn = creator args [ returns ] [ signals ]
  routine_body
  end idn

handler ::= idn = handler args [ returns ] [ signals ]
  routine_body
  end idn

routine_body ::= { equate }
  { own_var }
  { statement }

parms ::= [ parm, ... ]

parm ::= idn, ... : type
  | idn, ... : type_spec

args ::= ( [ decl, ... ] )

decl ::= idn, ... : type_spec

returns ::= returns ( type_spec, ... )

yields ::= yields ( type_spec, ... )

signals ::= signals ( exception, ... )

exception ::= name [ ( type_spec, ... ) ]
I Syntax

opidn ::= idn
       | transmit

where ::= where restriction, ...

restriction ::= idn has oper_decl, ...
              | idn in type_set

type_set ::= ( idn | idn has oper_decl, ... { equate } )
           | idn
           | reference $ name

oper_decl ::= name, ..., : type_spec
            | transmit

corstant ::= expression
            | type_spec

state_decl ::= [ stable ] decl
              | [ stable ] idn : type_spec := expression
              | [ stable ] decl, ..., := call

equate ::= idn = constant
          | idn = type_set
          | idn = reference

own_var ::= own decl
          | own idn : type_spec := expression
          | own decl, ..., := call [ @ primary ]
statement ::= decl
           | decl , ... := call [ @ primary ]
           | idn , ... := call [ @ primary ]
           | idn , ... := expression , ... 
           | primary . name := expression 
           | primary [ expression ] := expression 
           | call [ @ primary ] 
           | fork call 
           | seize expression do body end 
           | pause 
           | terminate 
           | enter_stmt 
           | coenter coarm { coarm } end
           | [ abort ] leave 
           | while expression do body end 
           | for_stmt 
           | if_stmt 
           | tagcase_stmt 
           | tagtest_stmt 
           | tagwait_stmt 
           | [ abort ] return [ ( expression , ... ) ]
           | yield [ ( expression , ... ) ]
           | [ abort ] signal name [ ( expression , ... ) ]
           | [ abort ] exit name [ ( expression , ... ) ]
           | [ abort ] break 
           | [ abort ] continue 
           | begin body end 
           | statement [ abort ] resignal name , ... 
           | statement except { when_handler }
           | [ others_handler ] 
           | end 

enter_stmt ::= enter topaction body end 
              | enter action body end
coarm ::= armtag [ foreach decl, ... in call ] body

armtag ::= action
        | topaction
        | process

for_stmt ::= for [ decl, ... ] in call do body end
          | for [ idn, ... ] in call do body end

if_stmt ::= if expression then body
          { elseif expression then body }
          [ else body ]
          end

tagcase_stmt ::= tagcase expression
                tag_arm { tag_arm }
                [ others : body ]
                end

tagtest_stmt ::= tagtest expression
                atag_arm { atag_arm }
                [ others : body ]
                end

tagwait_stmt ::= tagwait expression
                atag_arm { atagArm }
                end

tag_arm ::= tag name, ..., [ ( idn : type_spec ) ] : body
atag_arm ::= tag kind name, ..., [ ( idn : type_spec ) ] : body

tag_kind ::= tag
        | wtag

when_handler ::= when name, ..., [ ( decl, ... ) ] : body
                when name, ..., ( * ) : body

others_handler ::= others [ ( idn : type_spec ) ] : body

body ::= { equate }
        { statement }
type_spec ::= null
| node
| bool
| int
| real
| char
| string
| any
| image
| rep
| cvt
| sequence [ type_actual ]
| array [ type_actual ]
| atomic_array [ type_actual ]
| struct [ field_spec , ... ]
| record [ field_spec , ... ]
| atomic_record [ field_spec , ... ]
| oneof [ field_spec , ... ]
| variant [ field_spec , ... ]
| atomic_variant [ field_spec , ... ]
| proctype ( [ type_spec , ... ] ) [ returns ] [ signals ]
| itertype ( [ type_spec , ... ] ) [ yieled ] [ signals ]
| creatortype ( [ type_spec , ... ] ) [ returns ] [ signals ]
| handlertype ( [ type_spec , ... ] ) [ returns ] [ signals ]
| mutex [ type_actual ]
| reference
field_spec ::= name , ... : type_actual
reference ::= id
| idn [ actual_parm , ... ]
| reference $ name
actual_parm ::= constant
| type_actual
type_actual ::= type_spec [ with ( where opbinding , ... ) ]
opbinding ::= name , ... : primary
| expression | ::= | primary |
|           |     | call @ primary |
|           |     | ( expression ) |
|           |     | ~ expression   % 6 (precedence) |
|           |     | ~ expression   % 6 |
|           |     | expression ** expression % 5 |
|           |     | expression // expression % 4 |
|           |     | expression / expression % 4 |
|           |     | expression * expression % 4 |
|           |     | expression % 3 |
|           |     | expression + expression % 3 |
|           |     | expression - expression % 3 |
|           |     | expression < expression % 2 |
|           |     | expression <= expression % 2 |
|           |     | expression = expression % 2 |
|           |     | expression >= expression % 2 |
|           |     | expression == expression % 2 |
|           |     | expression == expression % 2 |
|           |     | expression -= expression % 2 |
|           |     | expression -= expression % 2 |
|           |     | expression += expression % 2 |
|           |     | expression += expression % 2 |
|           |     | expression && expression % 1 |
|           |     | expression && expression % 1 |
|           |     | expression | expression % 0 |
|           |     | expression || expression % 0 |
|           |     | expression || expression % 0 |

| primary | ::= | entity |
|         |     | call |
|         |     | primary . name |
|         |     | primary [ expression ] |

| call | ::= | primary ( [ expression , ... ] ) |
entity ::= nil
| true
| false
| int_literal
| real_literal
| char_literal
| string_literal
| self
| reference
| entity . name
| entity [ expression ]
| bind entity ([ bind_arg, ... ])
| type_spec $ { field, ... }
| type_spec $ [ [ expression : ] [ expression, ... ] ]
| type_spec $ name [ [ actual_parm, ... ] ]
| up ( expression )
| down ( expression )

field ::= name, ... : expression

bind_arg ::= *
| expression
Comment: a sequence of characters that begins with a percent sign (%), ends with a newline character, and contains only printing ASCII characters and horizontal tabs in between.

Separator: a blank character (space, vertical tab, horizontal tab, carriage return, newline, form feed) or a comment. Zero or more separators may appear between any two tokens, except that at least one separator is required between any two adjacent non-self-terminating tokens: reserved words, identifiers, integer literals, and real literals.

Reserved word: one of the identifiers appearing in bold face in the syntax. Upper and lower case letters are not distinguished in reserved words.

Name, idn: a sequence of letters, digits, and underscores that begins with a letter or underscore, and that is not a reserved word. Upper and lower case letters are not distinguished in names and ids.

Int_literal: a sequence of one or more decimal digits (0-9) or a backslash (\) followed by any number of octal digits (0-7) or a backslash and a sharp sign (#) followed by any number of hexadecimal digits (0-9, A-F in upper or lower case).

Real_literal: a mantissa with an (optional) exponent. A mantissa is either a sequence of one or more decimal digits, or two sequences (one of which may be empty) joined by a period. The mantissa must contain at least one digit. An exponent is 'E' or 'e', optionally followed by '+' or '-', followed by one or more decimal digits. An exponent is required if the mantissa does not contain a period.

Char_literal: a character representation other than single quote, enclosed in single quotes. A character representation is either a printing ASCII character (octal value 40 through 176) other than backslash, or an escape sequence consisting of a backslash (\) followed one to three printing characters as shown in Table 6-1 or Table I-1 below.

String_literal: a sequence of zero or more character representations other than double quote, enclosed in double quotes.

Table I-1 shows most of the character literals supported by Argus, except for the higher numbered octal escape sequences. For each character, the corresponding octal literal, hexadecimal literal, and normal literal(s) are shown. Upper or lower case letters may be used in escape sequences of the form \W*, \W, \W, \W, \W, \W, \W, \W, and \W. Note that an implementation need not support 256 characters, in which case only a subset of the literals listed will be legal.
<table>
<thead>
<tr>
<th>Character Escape Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>\000 ' \004 ' \001 ' \002 '</td>
</tr>
<tr>
<td>\010 ' \014 ' \012 ' \013 '</td>
</tr>
<tr>
<td>\017 ' \019 ' \020 ' \021 '</td>
</tr>
<tr>
<td>\022 ' \024 ' \023 ' \025 '</td>
</tr>
<tr>
<td>\026 ' \027 ' \030 ' \033 '</td>
</tr>
<tr>
<td>\036 ' \038 ' \040 ' \044 '</td>
</tr>
<tr>
<td>\046 ' \047 ' \050 ' \055 '</td>
</tr>
<tr>
<td>\057 ' \115 '</td>
</tr>
<tr>
<td>000</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>001</td>
</tr>
<tr>
<td>002</td>
</tr>
<tr>
<td>003</td>
</tr>
<tr>
<td>004</td>
</tr>
<tr>
<td>005</td>
</tr>
<tr>
<td>006</td>
</tr>
<tr>
<td>007</td>
</tr>
<tr>
<td>070</td>
</tr>
<tr>
<td>071</td>
</tr>
<tr>
<td>072</td>
</tr>
<tr>
<td>073</td>
</tr>
<tr>
<td>074</td>
</tr>
<tr>
<td>075</td>
</tr>
<tr>
<td>076</td>
</tr>
<tr>
<td>077</td>
</tr>
</tbody>
</table>
Appendix II
Built-in Types and Type Generators

The following sections specify the built-in types and the types produced by the built-in type generators of Argus. For each type and for each instance of each type generator, the objects of the type are characterized, and all of the operations of the type are defined. (An implementation may provide additional operations on the built in types, as long as these are operations that could be implemented in terms of those described in this section.)

All the built-in types (except for any) are transmissible. All instances of the built-in type generators (except for proctype and itertype) are transmissible if all their type parameters are transmissible. Transmission of the built-in types preserves value equality, except for objects of type real. However, in a homogeneous environment, reals can be transmitted without approximations. In a homogeneous environment, the only possible encode or decode failures are exceeding the representation limits of an Image, mutating the size of an array or atomic__array while it is being encoded or decoded, and improper decoding of cyclic objects (see Section 14.4).

All operations are indivisible except at calls to subsidiary operations (such as Int$similar within array[Int]$similar), at yields, and while waiting for locks.

The specifications given below are informal and are adapted from the book Abstraction and Specification in Program Development (Liskov, B. and Guttag, J., MIT Press, 1986). A specification starts out by giving a list of the operations and declarations of any formal parameters for the type. This is followed by an overview, which gives an introduction to the type and if necessary defines a way of describing the type’s objects and their values. Following this the individual operations are described. For each operation there is a heading and a statement of the operation’s effects. In the heading, the return values may be given names. The effects section describes the normal and exceptional behavior of the operation. The effects given are abstract, that is they are described using the vocabulary (or model) defined in the overview section. For example, objects of type Int are described using mathematical integers. Thus arithmetic expressions and comparisons used in defining Int operations are to be computed over the domain of mathematical integers.

An operation that (abstractly) mutates one of its arguments lists the arguments that it mutates in the clause following the word modifies. An operation is not allowed to mutate any objects, except for those listed in the modifies clause. (For the built-in mutable atomic type generators, modification only refers to the sequential state; it does not refer to changes in the locking information kept for each object.) When an argument, say a, is mutated, it is often necessary to describe its state at the start of the call as well as its final state at the end of the call. We use the notation \( a_{pre} \) for a’s state at the start of the call and the notation \( a_{post} \) for its state at the end of the call.

Some operations of the built-in type generators are only defined if the type generator is passed appropriate actual routine parameters (see Section 12.6). For example, the copy operation of the array
type generator, is only defined if there is an actual parameter passed (explicitly or implicitly) for the type
generator's copy operation. Thus array[int]$copy is defined but array[any]$copy is not defined. These
requirements are stated in a requires clause that precedes the description of the operation's effect. The
type of the expected routine is also described; remember that the actual operation parameter can have
fewer signals (see Section 6.1 and Section 12.8).

By convention, the order in which exceptions are listed in the operation type is the order in which the
various conditions are checked.

Operations with the same semantics (for example, null$equal and null$similar) or that can be
described in the same way (for example, int$add and int$sub) are grouped together to save space.

In defining the built-in types, we do not depend on users satisfying any constraints beyond those that
can be type-checked. This decision leads to more complicated specifications. For example, the behavior
of the elements iterator for arrays is defined even when the loop modifies the array.

II.1. Null

null = data type is copy, equal, similar, transmit

Overview

The type null has exactly one, immutable, atomic object, represented by the literal nil. Nil is
generally used as a placeholder in type definitions using oneofs or variants.

Operations

equal = proc (n1, n2: null) returns (bool)
similar = proc (n1, n2: null) returns (bool)
    effects Returns true.

copy = proc (n: null) returns (null)
transmit = proc (n: null) returns (null)
    effects Returns nil.

II.2. Nodes

node = data type is here, copy, equal, similar, transmit

Overview

Objects of type node are immutable and atomic, and stand for physical nodes. Implementations
should provide some mechanism for translating a node "address" into a node object and vice
versa. (However, these do not have to be operations of type node.)

Operations

here = proc () returns (node)
    effects Returns the node object for the caller's node.

equal = proc (n1, n2: node) returns (bool)
similar = proc (n1, n2: node) returns (bool)
    effects Returns true if and only if n1 and n2 are the same node.
II.3. Booleans

In the language, the data type `bool` is used to denote logical truth values. The two immutable, atomic objects of type `bool` are `true` and `false`, which represent logical truth values.

The language also provides the operators `and`, `or`, `not`, `equal`, `similar`, `copy`, and `transmit` for conditional evaluation of boolean expressions. See Section 9.15 for details.

**Operations**

- `and` takes two boolean values and returns `true` if both operands are true; otherwise, it returns `false`.
- `or` takes two boolean values and returns `true` if either or both operands are true; otherwise, it returns `false`.
- `not` takes a boolean value and returns `false` if the operand is true; otherwise, it returns `true`.
- `equal` and `similar` take two boolean values and return `true` if both operands are true or both are false; otherwise, they return `false`.
- `copy` takes a boolean value and returns a new boolean value.
- `transmit` takes a boolean value and returns a new boolean value.

II.4. Integers

The `int` data type includes numeric operations such as addition, subtraction, multiplication, and division. Integers are used to represent whole numbers.

**Overview**

Objects of type `int` are immutable and atomic, and are intended to model a subrange of the mathematical integers. The exact range is not part of the language definition and can vary somewhat from implementation to implementation. The language is designed to provide a closed interval [MININT, MAXINT], where MININT and MAXINT are positive integers, and the number of characters — see Section 11.4.6 — is an indication of the actual number of characters in the result of the operation if the result would be outside this interval. See Appendix F for implementation details.

**Operations**

- `add` takes two integer values and returns their sum.
- `sub` takes two integer values and returns the difference.
- `mul` takes two integer values and returns their product.
- `div` takes two integer values and returns the quotient.
- `rem` takes two integer values and returns the remainder.
- `max` takes two integer values and returns the maximum.
- `min` takes two integer values and returns the minimum.
- `eqv` takes two integer values and returns `true` if both values are equal; `false` otherwise.
- `neqv` takes two integer values and returns `true` if both values are not equal; `false` otherwise.
- `less` takes two integer values and returns `true` if the first value is less than the second; `false` otherwise.
- `leqv` takes two integer values and returns `true` if the first value is less than or equal to the second; `false` otherwise.
- `greater` takes two integer values and returns `true` if the first value is greater than the second; `false` otherwise.
- `geqv` takes two integer values and returns `true` if the first value is greater than or equal to the second; `false` otherwise.
- `abs` takes an integer value and returns its absolute value.
- `sign` takes an integer value and returns its sign (1 for positive, 0 for zero, and -1 for negative).
- `two` takes an integer value and returns an integer value that is twice the input value.
- `half` takes an integer value and returns an integer value that is half the input value.
- `power` takes two integer values and returns the result of raising the first value to the power of the second value.
- `log` takes an integer value and returns its logarithm.
- `sqrt` takes an integer value and returns its square root.
- `round` takes an integer value and returns the nearest integer value.
- `floor` takes an integer value and returns the largest integer value that is less than or equal to the input value.
- `ceil` takes an integer value and returns the smallest integer value that is greater than or equal to the input value.
- `trunc` takes an integer value and returns the integer value that is closest to the input value.
- `signif` takes an integer value and returns the number of significant digits in the input value.
- `frac` takes an integer value and returns the fractional part of the input value.
- `integer` takes an integer value and returns the integer value that is closest to the input value.
- `float` takes an integer value and returns the floating-point value that is closest to the input value.
- `real` takes an integer value and returns the real value that is closest to the input value.
- `imag` takes an integer value and returns the imaginary part of the input value.
minus = proc (x: int) returns (int) signals (overflow)
effects Returns the negative of x; signals overflow if the result would lie outside the represented interval.

div = proc (x, y: int) returns (q; r: int) signals (zero_divide, overflow)
effects Signals zero_divide if y = 0. Otherwise returns the integer quotient of dividing x by y; that is, \( x = q \cdot y + r \), for some integer \( r \) such that \( 0 \leq r < |y| \). Signals overflow if \( q \) would lie outside the represented interval.

mod = proc (x, y: int) returns (r: int) signals (zero_divide, overflow)
effects Signals zero_divide if y = 0. Otherwise returns the integer remainder of dividing x by y; that is, \( r \) is such that \( 0 \leq r < |y| \), for some integer \( q \) \( x = q \cdot y + r \). Signals overflow if \( r \) would lie outside the represented interval.

power = proc (x, y: int) returns (x: int) signals (negative, exponent, overflow)
effects Signals negative, exponent if \( y < 0 \). Computes \( x^y \); signals overflow if the result would lie outside the represented interval. \( x^0 \) = 1 by definition.

abs = proc (x: int) returns (int) signals (overflow)
effects Returns the absolute value of x; signals overflow if the result would lie outside the represented interval.

from_to_by = iter (from, to, by: int) yields (int)
effects Yields the integers from from to to, incrementing by by each time, that is, yields from, from+by, ..., from+by*(n-1) where n is the smallest integer such that from+by<n*by ≤ to. If by = 0, then yields from iterations. Otherwise yields \( \min(\text{to}, \text{to} \cdot \text{by}) \) to end by by > 0, or if from < to end by by < 0. This function is identical to \( \text{from}_\text{to}_\text{by}(\text{from}, \text{to}, 1) \).

from_to = iter (from, to: int) yields (int)
effects Yields The effect is identical to \( \text{from}_\text{to}_\text{by}(\text{from}, \text{to}, 1) \).

max = proc (x, y: int) returns (int)
effects If \( x \geq y \), then returns x; otherwise returns y.

min = proc (x, y: int) returns (int)
effects If \( x \leq y \), then returns x; otherwise returns y.

parse = proc (x: string) returns (int) signals (bad, format, overflow)
effects \( S \) must be an integer that can be stored in an \( \text{int} \); \( S \) is an optional leading plus or minus sign; \( S \) is not out of the range \([-2^{31}, 2^{31}-1] \). Otherwise returns the integer corresponding to \( x \); signals overflow if the result would lie outside the represented interval.

unparse = proc (x: int) returns (string)
effects Produces the string representing the integer value of \( x \) in decimal notation, preceded by a minus sign if \( x < 0 \). Leading zeros are suppressed, and there is no leading plus sign for positive integers.

lt = proc (x, y: int) returns (true)
gt = proc (x, y: int) returns (true)
lge = proc (x, y: int) returns (true)
gge = proc (x, y: int) returns (true)
effects These are the standard ordering relations.

equal = proc (x, y: int) returns (true)
similar = proc (x, y: int) returns (true)
effects Returns true if \( x \) and \( y \) are the same integer; returns false otherwise.

copy = proc (x: int) returns (int)
effects Returns x.
transmit = proc (x: int) returns (y: int) signals(failure(string))
effects Returns \( y \) such that \( x = y \) or signals \( \text{failure} \) if \( x \) cannot be represented in the
implementation on the receiving end.

II.5. Reals

real = data type is add, sub, minus, mul, div, power, abs, max, min, exponent, mantissa, l2r, r2l,
trunc, parse, unparse, it, le, ge, gt, equal, similar, copy, transmit

Overview

The type real models a subset of the mathematical numbers. It is used for approximate or floating
point arithmetic. Reals are immutable and atomic, and are written as a mantissa with an optional
exponent. See Appendix I for the format of real literals.

Each implementation represents a subset of the real numbers in:
\[
D = \{-\text{real\_max}, \text{real\_min}\} \cup \{0\} \cup \{\text{real\_min}, \text{real\_max}\}
\]
where
\[
0 < \text{real\_min} < 1 < \text{real\_max}
\]
Numbers in \( D \) are approximated by the implementation with a precision of \( p \) decimal digits such that:
- \( \forall r \in D \Rightarrow \text{Approx}(r) \in \text{Real} \)
- \( \forall r \in \text{Real} \Rightarrow \text{Approx}(r) = r \)
- \( \forall r \in D - \{0\} \Rightarrow |(\text{Approx}(r) - r)/r| < 10^{-p} \)
- \( \forall r, s \in D \Rightarrow r \leq s \Rightarrow \text{Approx}(r) \leq \text{Approx}(s) \)
- \( \forall r \in D \Rightarrow \text{Approx}(\neg r) = \neg \text{Approx}(r) \)

We define \( \text{Max\_width} \) and \( \text{Exp\_width} \) to be the smallest integers such that every nonzero element
of real can be represented in "standard" form (exactly one digit, not zero, before the decimal
point) with no more than \( \text{Max\_width} \) digits of mantissa and no more than \( \text{Exp\_width} \) digits of
exponent.

Real operations signal an exception if the result of a computation lies outside of \( D \); overflow
occurs if the magnitude exceeds real\_max, and underflow occurs if the magnitude is less than
real\_min.

Operations

add = proc (x, y: real) returns (real) signals (overflow, underflow)
effects Computes the sum \( z \) of \( x \) and \( y \); signals overflow or underflow if \( z \) is outside of \( D \), as
explained earlier. Otherwise returns an approximation such that:
- \( (x, y \geq 0 \lor x, y \leq 0) \Rightarrow \text{add}(x, y) = \text{Approx}(x + y) \)
- \( \text{add}(x, y) = (1 + e)(x + y) \quad |e| < 10^{-p} \)
- \( \text{add}(x, 0) = x \)
- \( \text{add}(x, y) = \text{add}(y, x) \)
- \( x \leq x' \Rightarrow \text{add}(x, y) \leq \text{add}(x', y) \)

sub = proc (x, y: real) returns (real) signals (overflow, underflow)
effects Computes \( x - y \); the result is identical to \( \text{add}(x, -y) \).

minus = proc (x: real) returns (real)
effects Returns \( -x \).

mul = proc (x, y: real) returns (real) signals (overflow, underflow)
effects Returns \( \text{approx}(x \cdot y) \); signals overflow or underflow if \( x \cdot y \) is outside of \( D \).

div = proc (x, y: real) returns (real) signals (zero\_divide, overflow, underflow)
effects If \( y = 0 \), signals \( \text{zero\_divide} \). Otherwise returns \( \text{approx}(x/y) \); signals overflow or
underflow if \( x/y \) is outside of \( D \).
power = proc (x, y: real) returns (real)
    signals (zero_divide, complex_result, overflow, underflow)
    Effects if x = 0 and y < 0, signals zero_divide. If x < 0 and y is nonintegral, signals
    complex_result. Otherwise returns an approximation to \(x^y\), good to \(p\) significant digits;
    signals overflow or underflow if \(x^y\) is outside of \(D\).

abs = proc (x: real) returns (real)
    Effects Returns the absolute value of \(x\).

max = proc (x, y: real) returns (real)
    Effects if \(x \geq y\), then returns \(x\), otherwise returns \(y\).

min = proc (x, y: real) returns (real)
    Effects if \(x \leq y\), then returns \(x\), otherwise returns \(y\).

exponent = proc (x: real) returns (int) signals (undefined)
    Effects if \(x = 0\), signals undefined. Otherwise returns the exponent that would be used in
    representing \(x\) as a literal in standard form, that is, returns
    \(
    \text{max}\{\lfloor \log |x| \rfloor \text{ and } 0\} \text{ if } |x| \leq 10^0\}
    \)

mantissa = proc (x: real) returns (real)
    Effects Returns the mantissa of \(x\) when represented in standard form, that is, returns
    \(\text{approx}(x/10^e)\), where \(e = \text{exponent}(x)\). If \(x = 0.0\), returns 0.0.

i2r = proc (i: int) returns (real) signals (overflow)
    Effects Returns \(\text{approx}(i)\); signals overflow if \(i\) is not in \(D\).

r2i = proc (x: real) returns (int) signals (overflow)
    Effects Rounds \(x\) to the nearest integer and toward zero in case of a tie. Signals overflow if
    the result lies outside the represented range of integers.

trunc = proc (x: real) returns (int) signals (overflow)
    Effects Truncates \(x\) toward zero; signals overflow if the result would be outside the
    represented range of integers.

parse = proc (s: string) returns (real) signals (bad_format, overflow, underflow)
    Effects Returns \(\text{approx}(z)\), where \(z\) is the value represented by the string \(s\) (see Appendix I).
    \(S\) must represent a real or integer literal with an optional leading plus or minus sign;
    otherwise signals bad_format. Signals underflow or overflow if \(z\) is not in \(D\).

unparse = proc (x: real) returns (string)
    Effects Returns a real literal such that \(\text{parse(unparse}(x)) = x\). The general form of the literal
    is:

    \([-\) i_field.f_field [e \pm x_field ] \]

    Leading zeros in i_field and trailing zeros in f_field are suppressed. If \(x\) is integral and
    within the range of represented integers, then f_field and the exponent are not present. If
    \(x\) can be represented by a mantissa of no more than Max_width digits and no exponent
    (that is, \(-1 \leq \text{exponent}(\text{any}) < \text{Max_width}\)), then the exponent is not present.
    Otherwise the literal is in standard form, with Exp_width digits of exponent.

lt = proc (x, y: real) returns (bool)
le = proc (x, y: real) returns (bool)
ge = proc (x, y: real) returns (bool)
gt = proc (x, y: real) returns (bool)
    Effects These are the standard ordering relations.

equal = proc (x, y: real) returns (bool)
similar = proc (x, y: real) returns (bool)
    Effects Returns true if \(x\) and \(y\) are the same number; returns false otherwise.
copy = proc (x: real) returns (real)
  effects Returns x.

transmit = proc (x: real) returns (real) signals (false, false)
  effects Returns output(x) where output is the transmission function for the receiving
  implementation of Auger or signal's interface and cannot be represented on the receiving
  end.

II.6. Characters

cchar = data type ia i2c, c2l, lt, le, ge, gl, equal, similar, copy, transmit

Overview

Type cchar provides the alphabet for text manipulation. Characters are treatable and atomic, and
form an ordered set. Every implementation must contain at least 128, but no more than 512,
characters; the first 128 characters are the same across all implementations, as standard order.

Operations i2c and c2l convert between ints and chars (using the ASCII encoding for the first 128
characters). The smallest character corresponds to 0, and characters are numbered sequentially up to char_{127} the highest, corresponding to the highest character. This numbering
determines the ordering of the characters.

Printing ASCII characters (each cchar contains at most 170, other than single quote or backslash, can
be written as that character enclosed by quotes in the source. See Appendix I for the syntax of character
literals and tables of character escape sequences.

Operations

i2c = proc (x: int) returns (cchar) signals (false, false)
  effects Returns the character corresponding to x; signals illegal_char if x is not in the range
  [0, char_{127}].

c2l = proc (c: cchar) returns (int)
  effects Returns the integer corresponding to c (using the ASCII encoding if c is an ASCII
  character).

lt = proc (c1, c2: cchar) returns (false)
lte = proc (c1, c2: cchar) returns (true)
ge = proc (c1, c2: cchar) returns (false)
gte = proc (c1, c2: cchar) returns (true)

equal = proc (c1, c2: cchar) returns (true)
similar = proc (c1, c2: cchar) returns (true)

effects Returns true if c1 and c2 are the same character, i.e., return (c2(c1) = c1(c2)).

copy = proc (c1: cchar) returns (cchar)
  effects Returns c1.

transmit = proc (c1: cchar) returns (cchar) signals (false, false)
  effects Returns c1. Signals illegal if c1 is not representable by the implementation on
  the receiving end.
II.7. Strings

`string = data type is c2s, concat, append, substr, rest, size, empty, fetch, chars, indexes, indexc, s2ac, ac2s, s2sc, sc2s, lt, le, ge, gt, equal, similar, copy, transmit`

Overview

Type `string` is used for representing text. A string is an immutable and atomic tuple of zero or more characters. The characters of a string are indexed sequentially starting from one. Strings are lexicographically ordered based on the ordering for characters.

A string literal is written as a sequence of zero or more character representations enclosed in double quotes. See Appendix I for a description of the character escape sequences that can be used within string literals. No string can have a size greater than `int_max`; however, an implementation may restrict string lengths to a value less than `int_max`. If the result of a string operation would be a string containing more than the maximum number of characters, the operation signals `limits`.

Operations

`c2s = proc (c: char) returns (string)`  
effects Returns a string containing `c` as its only character.

`concat = proc (s1, s2: string) returns (r: string) signals (limits)`  
effects Returns the concatenation of `s1` and `s2`. That is, `r[i]=s1[i]` for `i` an index of `s1` and `r[size(s1)+i]=s2[i]` for `i` an index of `s2`. Signals `limits` if `r` would be too large for the implementation.

`append = proc (s: string, c: char) returns (r: string) signals (limits)`  
effects Returns a new string having the characters of `s` in order followed by `c`. That is, `r[size(s)+1]=c`. Signals `limits` if the new string would be too large for the implementation.

`substr = proc (s: string, at: int, cnt: int) returns (string) signals (bounds, negative size)`  
effects If `cnt < 0`, signals `negative size`. If `at < 1` or `at > size(s)+1`, signals `bounds`. Otherwise returns a string having the characters `s[at]`, `s[at+1]`, ... in that order; the new string contains `min(cnt, size(at)+1)` characters. For example,

- `substr ("abcdef", 2, 3) = "bod"`
- `substr ("abcdef", 2, 7) = "bodfe"`
- `substr ("abcdef", 7, 1) = ""`

Note that if `min(cnt, size-at+1) = 0`, `substr` returns the empty string.

`rest = proc (s: string, i: int) returns (r: string) signals (bounds)`  
effects Signals `bounds` if `i < 0` or `i > size(s) + 1`; otherwise returns a string whose first character is `s[i]`, whose second is `s[i+1]`, ..., and whose `size(i)`th character is `s[size(s)]`. Note that if `i = size(s)+1`, `rest` returns the empty string.

`size = proc (s: string) returns (int)`  
effects Returns the number of characters in `s`.

`empty = proc (s: string) returns (bool)`  
effects Returns true if `s` is empty (contains no characters); otherwise returns false.

`fetch = proc (s: string, i: int) returns (char) signals (bounds)`  
effects Signals `bounds` if `i < 0` or `i > size(s)`; otherwise returns the `i`th character of `s`.

`chars = iter (s: string) yields (char)`  
effects Yields, in order, each character of `s` (i.e., `s[1]`, `s[2]`, ...).
indexes = proc (s1, s2: string) returns (int)
effects If s1 occurs as a substring in s2, returns the least index at which s1 occurs. Returns 0 if s1 does not occur in s2, and 1 if s1 is the empty string. For example,
  indexes("abc", "abcabc") = 1
  indexes("bc", "abcabc") = 2
  indexes("", "abcde") = 1
  indexes("bcb", "abcde") = 0

indexc = proc (c: char, s: string) returns (int)
effects If c occurs in s, returns the least index at which c occurs; returns 0 if c does not occur in s.

s2ac = proc (s: string) returns (array[char])
effects Stores the characters of s as elements of a new array of characters, a. The low bound of the array is 1, the size is size(s), and the i\textup{th} element of the array is the i\textup{th} character of s, for 1 \leq i \leq size(s).

ac2s = proc (a: array[char]) returns (string)
effects This is the inverse of s2ac. The result is a string with characters in the same order as in a. That is, the i\textup{th} character of the string is the (i-array[char]$\text{low}(a)-1)$th element of a.

s2sc = proc (s: string) returns (sequence[char])
effects Transforms a string into a sequence of characters. The size of the sequence is size(s). The i\textup{th} element of the sequence is the i\textup{th} character of s, for 1 \leq i \leq size(s).

sc2s = proc (s: sequence[char]) returns (string)
effects This is the inverse of s2sc. The result is a string with characters in the same order as in s. That is, the i\textup{th} character of the string is the i\textup{th} element of s.

lt = proc (s1, s2: string) returns (bool)
le = proc (s1, s2: string) returns (bool)
ge = proc (s1, s2: string) returns (bool)
gt = proc (s1, s2: string) returns (bool)
effects These are the usual lexicographic ordering relations on strings, based on the ordering of characters. For example,
  "abc" < "aca"
  "abc" < "abca"

equal = proc (s1, s2: string) returns (bool)
similar = proc (s1, s2: string) returns (bool)
effects Returns true if s1 and s2 are the same string; otherwise returns false.

copy = proc (s1: string) returns (string)
effects Returns s1.

transmit = proc (s1: string) returns (string) signals (failure(string))
effects Returns s1. Signals failure only if s1 is not representable on the receiving end.
II.8. Sequences

sequence = data type [t: type] is new, e2s, fill, fill_copy, replace, addl, addh, remh, reml, concat,
subseq, size, empty, fetch, bottom, top, elements, indexes, a2s, s2a,
equal, similar, copy, transmit

Overview

Sequences represent immutable tuples of objects of type t. The elements of the sequence can be
indexed sequentially from 1 up to the size of the sequence. Although a sequence is immutable,
the elements of the sequence can be mutable objects. The state of such mutable elements may
change; thus, a sequence object is atomic only if its elements are also atomic.

Sequences can be created by calling sequence operations and by means of the sequence
constructor, see Section 6.2.8.

Any operation call that attempts to access a sequence with an index that is not within the defined
range terminates with the bounds exception. The size of a sequence can be no larger than the
largest positive int (int_max), but an implementation may restrict sequences to a smaller upper
bound. An attempt to construct a sequence which is too large results in a limits exception.

Operations

new = proc () returns (sequence[t])
  effects Returns the empty sequence.

e2s = proc (elem: t) returns (sequence[t])
  effects Returns a one-element sequence having elem as its only element.

fill = proc (cnt: int, elem: t) returns (sequence[t]) signals (negative_size, limits)
  effects If cnt < 0, signals negative_size. If cnt is larger than the maximum sequence size
  supported by the implementation, signals limits. Otherwise returns a sequence having
  cnt elements each of which is elem.

fill_copy = proc (cnt: int, elem: t) returns (sequence[t])
  signals (negative_size, limits, failure(string))
  requires t has copy: proctype (t) returns (t) signals (failure(string))
  effects If cnt < 0, signals negative_size. If cnt is bigger than the maximum size of
  sequences that the implementation supports, signals limits. Otherwise returns a new
  sequence having cnt elements each of which is a copy of elem, as made by $copy. Note
  that $copy is called cnt times. Any failure signal raised by $copy is immediately
  resignalized. This operation does not originate any failure signals by itself.

replace = proc (s: sequence[t], i: int, elem: t) returns (sequence[t]) signals (bounds)
  effects If i < 1 or i > high(s), signals bounds. Otherwise returns a sequence with the same
  elements as s, except that elem is in the ith position. For example,
  replace(sequence[int][2,5], 1, 6) = sequence[int][6,5]

addl = proc (s: sequence[t], elem: t) returns (r: sequence[t]) signals (limits)
  effects Returns a sequence with the same elements as s followed by one additional
  element, elem. That is, i = s[i] for i an index of s, and i = size(s)+1 = elem. If the resulting
  sequence would be larger than the implementation supports, signals limits.

addh = proc (s: sequence[t], elem: t) returns (r: sequence[t]) signals (limits)
  effects Returns a sequence having elem as the first element followed by the elements of s
  in order. That is, i = elem and i = s[i-1] for i = 2, ..., size(r). If the resulting sequence
  would be larger than the implementation supports, signals limits.

remh = proc (s: sequence[t]) returns (r: sequence[t]) signals (bounds)
  effects If s is empty, signals bounds. Otherwise returns a sequence having all elements of s
  in order, except the last one. That is, size(r) = size(s)−1 and r[i] = s[i] for i = 1, ..., size(s)−1.
reml = proc (a: sequence[0]) returns (r: sequence[0]) signals (timeout)
effects if a is empty, signals timeout. Otherwise returns a sequence containing all elements of a in order, except the first one. That is, \( a[i] \) for \( i = 1, \ldots, \text{size}(a) - 1 \).

concat = proc (s1, s2: sequence[n]) returns (r: sequence[n]) signals (timeout)
effects Returns the concatenation of s1 and s2. A sequence having the elements of s1 followed by the elements of s2, that is, \( a[i] \) for \( i \) an index of s1 and \( a[i] \) for \( i \) an index of s2. Signals timeout if the resulting sequence would be larger than the implementation supports.

subseq = proc (a: sequence[n], st, end: int) returns (list)
    signals (timeout)
effects If \( \text{end} < 0 \), sets return to false. If \( \text{st} < 1 \) or \( \text{st} > \text{size}(a) + 1 \), signals timeout. Otherwise returns a sequence having the subsequence \( a[\text{st} - 1, \text{end}] \) in that order; the new sequence contains exactly \( n \) elements. \( n = \text{size}(a) - \text{size}(a) \).

size = proc (a: sequence[n]) returns (n)
effects Returns the number of elements in a.

empty = proc (a: sequence[n]) returns (bool)
effects Returns true if a contains no elements; otherwise returns false.

fetch = proc (a: sequence[n], i: int) returns (a[i]) signals (timeout)
effects If \( i < 1 \) or \( i > \text{size}(a) \), signals timeout. Otherwise returns the ith element of a.

bottom = proc (a: sequence[n]) returns (a[1]) signals (timeout)
effects If a is empty, signals timeout. Otherwise returns a[1].

top = proc (a: sequence[n]) returns (a[\text{size}(a)]) signals (timeout)
effects If a is empty, signals timeout. Otherwise returns a[\text{size}(a)].

elements = iter (a: sequence[n]) yields (i)
effects Yields the elements of a in order (i.e., a[1], a[2], ...).

indexes = iter (a: sequence[n]) yields (i)
effects Yields the indexes of a from 1 to size(a).

s2s = proc (a: array[n]) returns (sequence[n])
effects Returns a sequence having the elements of a in the same order as in a.

s2s = proc (a: sequence[n]) returns (array[n])
effects Returns a new array with lower bound 1 and having the elements of a in the same order as in a.

equal = proc (a1, a2: sequence[n]) returns (true, signals (timeout))
requires (true, signals (timeout))
effects Returns true if a1 and a2 are equal in the sense defined by isequal. The effect of this operation is: one of the following statements holds:
\[ a_1 = a_2 \]
\[ \text{a1[1]} = \text{a2[1]} \] and \[ \text{equal(a1[2:] a2[1:])} \]
\[ \text{equal(a1[1:] a2[1:])} \] and \[ \text{equal(a1[1:] a2[1:])} \]

return (true)

similar = proc (a1, a2: sequence[n]) returns (true, signals (timeout))
requires (true, signals (timeout))
effects Returns true if a1 and a2 are similar in the sense defined by issimilar. Similar works in the same way as equal, except that a1 and a2 are not required to be equal.
copy = proc (s: sequence[i]) returns (sequence[i]) signals (failure(string))
    requires t has copy: proc type (t) returns (t) signals (failure(string))
effects Returns a sequence having as elements copies of the elements of s. The effect is equivalent to that of the following procedure body:
    qt = sequence[t]
y := qt$new()
for e : t in qt$elements(s) do
    y := qt$addh(y, t$copy(e)) resignal failure
end
return (y)

transmit = proc (s: sequence[i]) returns (sequence[i]) signals (failure(string))
    requires t has transmit
    effects Returns a sequence having as elements transmitted copies of the elements of s in the same order. Sharing among elements is preserved. Signals failure if this cannot be represented on the receiving end and also resinals any failures from $transmit.

II.9. Arrays

array = data type [t: type] la create, new, predict, fill, fill_copy, addh, addl, remh, reml,
    set_low, trim, store, fetch, bottom, top, empty, size, low, high, elements, indexes,
    equal, similar, similar1, copy, copy1, transmit

Overview

Arrays are mutable objects that represent tuples of elements of type t that can grow and shrink dynamically. Each array's state consists of this tuple of elements and a low bound (or index). The elements are indexed sequentially, starting from the low bound. Each array also has an identity as an object.

Arrays can be created by calling array operations create, new, fill, fill_copy, and predict. They can also be created by means of the array constructor, which specifies the array low bound, and an arbitrary number of initial elements, see Section 6.2.9.

Operations low, high, and size return the current low and high bounds and size of the array. For array a, size(a) is the number of elements in a, which is zero if a is empty. These are related by the equation: high(a) = low(a) + size(a) - 1.

For any index i between the low and high bound of an array, there is a defined element, a[i]. The bounds exception is raised when an attempt is made to access an element outside the defined range. Any array must have a low bound, a high bound, and a size which are all legal integers. An implementation may restrict these to some smaller range of integers. A call that would lead to an array whose low or high bound or size is outside the defined range terminates with a limits exception.

Operations

create = proc (lb: int) returns (array[i]) signals (limits)
effects Returns a new, empty array with low bound lb. Limits occurs if the resulting array would not be supported by the implementation.

new = proc () returns (array[i])
effects Returns a new, empty array with low bound 1. Equivalent to create(1).
store = proc (a: array[t], i: int, elem: t) signals (bounds)
  modifies a.
  effects If i < low(a) or i > high(a), signals bounds; otherwise makes elem the element of a
  with index i.

fetch = proc (a: array[t], i: int) returns (t) signals (bounds)
  effects If i < low(a) or i > high(a), signals bounds; otherwise returns the element of a with
  index i.

bottom = proc (a: array[t]) returns (t) signals (bounds)
  effects If a is empty, signals bounds; otherwise returns a[low(a)].

top = proc (a: array[t]) returns (t) signals (bounds)
  effects If a is empty, signals bounds; otherwise returns a[high(a)].

empty = proc (a: array[t]) returns (bool)
  effects Returns true if a contains no elements; otherwise returns false.

size = proc (a: array[t]) returns (int)
  effects Returns a count of the number of elements of a.

low = proc (a: array[t]) returns (int)
  effects Returns the low bound of a.

high = proc (a: array[t]) returns (int)
  effects Returns the high bound of a.

elements = iter (a: array[t]) yields (t) signals (failure(string))
  effects Yields the elements of a, exactly once for each index, from the low bound to the high
  bound (i.e., bottom(a), ..., top(a)). The elements are fetched one at a time, using
  the indexes that were legal at the start of the call. If, during the iteration, a is modified so
  that fetching at a previously legal index signals bounds, then the iterator signals failure
  with the string "bounds". The iterator is divisible at yields.

indexes = iter (a: array[t]) yields (int)
  effects Yields the indexes of a from the low bound of a to the high bound of a. Note
  that indexes is unaffected by any modifications done by the loop body. It is divisible at
  yields.

equal = proc (a1, a2: array[t]) returns (bool)
  effects Returns true if a1 and a2 refer to the same array object; otherwise returns false.

similar = proc (a1, a2: array[t]) returns (bool) signals (failure(string))
  requires t has similar: proctype (t, t) returns (bool) signals (failure(string))
  effects Returns true if a1 and a2 have the same low and high bounds and if their elements
  are pairwise similar as determined by $similar. This effect of this operation is equivalent
  to the following procedure body (except that this operation is only divisible at calls to
  $similar):
  at = array[t]
  if at@low(a1) == at@low(a2) cor at@size(a1) == at@size(a2)
    then return (false)
  end
  for i: int in at@indexes(a1) do
    if ~($similar(a1[i], a2[i])) then return (false) end
  resignal failure
  except when bounds: signal failure("bounds") end
  end
  return (true)
II.9 Arrays

\[
\text{similar1} = \text{proc (a1, a2: array[t]) returns (bool) signals (failure(string))}
\text{requires f has equal: proctype (t, t) returns (bool) signals (failure(string))}
\text{effects Returns true if a1 and a2 have the same low and high bounds and if their elements}
\text{are pairwise equal as determined by \$equal. This operation works the same way as}
\text{\$similar, except that \$equal is used instead of \$similar.}
\]
\[
\text{copy} = \text{proc (a: array[t]) returns (b: array[t]) signals (failure(string))}
\text{requires f has copy: proctype (t) returns (t) signals (failure(string))}
\text{effects Returns a new array b with the same low and high bounds as a and such that each}
\text{element b[i] contains \$copy(a[i]). The effect of this operation is equivalent to the}
\text{following body (except that it is only divisible at \$calls to \$copy):}
\text{b: array[t]} \leftarrow \text{array[t]\$copy(a)}
\text{for i: int in array[t]\$indexes(a) do}
\text{b[i] \leftarrow \$copy(a[i])}
\text{resignal failure}
\text{except when bounds: signal failure("bounds") end}
\text{return (b)}
\]
\[
\text{copy1} = \text{proc (a: array[t]) returns (b: array[t])}
\text{effects Returns a new array b with the same low and high bounds as a and such that each}
\text{element b[i] contains the same element as a[i].}
\]
\[
\text{transmit} = \text{proc (a: array[t]) returns (b: array[t]) signals (failure(string))}
\text{requires f has transmit}
\text{effects Returns a new array b with the same low and high bounds as a and such that each}
\text{element b[i] contains a transmitted copy of a[i]. Sharing among the elements of a is}
\text{preserved in b. Signals failure if b cannot be represented on the receiving end or if}
\text{fetching an element at a legal index of a\_pre causes a bounds exception and resinals any}
\text{failure signals raised by \$transmit.}
\]

II.10. Atomic Arrays

\[
\text{atomic\_array = data type [t: type] is create, new, predict, fill, fill\_copy, addh, addi, remh, remi, set\_low, trim, store, fetch, bottom, top, empty, size, low, high, elements, indexes, a2a, a2aa, equal, similar, similar1, copy, copy1, transmit, test\_and\_read, test\_and\_write, can\_read, can\_write, read\_lock, write\_lock}
\]

Overview

Atomic\_arrays are mutable atomic objects that represent tuples of elements of type t that can
grow and shrink dynamically. Each atomic\_array\_s (sequential) state consists of this tuple
of elements and a low bound (or index). The elements are indexed sequentially, starting from
the low bound. Each atomic\_array also has an identity as an object.

Atomic\_arrays can be created by calling atomic\_array operations create, new, fill, fill\_copy, and
predict. They can also be created by means of the atomic\_array constructor, which specifies the
array low bound, and an arbitrary number of initial elements, see Section 6.2.9.

Operations low, high, and size return the current low and high bounds and size of the
atomic\_array. For an atomic\_array a, size(a) is the number of elements in a, which is zero if a is
empty. These are related by the equation: high(a) = low(a) + size(a) - 1.
For any index $i$ between the low and high bound of an atomic_array, there is a defined element, $a[i]$. The bounds exception is raised when an attempt is made to access an element outside the defined range. Any atomic_array must have a low bound, a high bound, and a size which are all legal integers. An implementation may restrict these to some smaller range of integers. A call that would lead to an atomic_array whose low or high bound or size is outside the defined range terminates with a limits exception. limits exception.

Atomic_arrays use read/write locking to achieve atomicity. The locking rules are described in Section 2.2.2. It is an error if a process that is not in an action attempts to test or obtain a lock; when this happens the guardian running the process will crash. As defined below, the only operation that (in the normal case) does not attempt to test or obtain a lock is the equal operation.

**Operations**

create = proc (lb: Int) returns (a: atomic_array[1]) signals (limits)
effects Returns a new, empty atomic_array $a$ with low bound $lb$. Limits occurs if the resulting atomic_array would not be supported by the implementation. The caller obtains a read lock on $a$.

ew = proc ( ) returns (atomic_array[1])
effects Equivalent to create(1).

predict = proc (lb, cnt: Int) returns (a: atomic_array[1]) signals (limits)
effects Returns a new, empty atomic_array $a$ with low bound $lb$. The caller obtains a read lock on $a$. This is essentially the same as create(lb), except that the absolute value of $cnt$ is a prediction of how many adds or adds are likely to be performed on this new atomic_array. If $cnt > 0$, adds are expected; otherwise addrs are expected. These operations may execute faster than if the atomic_array had been produced by calling create. Limits occurs if the resulting atomic_array would not be supported by the implementation because of its initial low bound (not because of its predicted size or because of the predicted high or low bound).

fill = proc (lb, cnt, elem: t) returns (atomic_array[1]) signals (negative_size, limits)
effects If $cnt < 0$, signals negative_size. Returns a new atomic_array with low bound $lb$ and size $cnt$, and with $elem$ as each element; if this new atomic_array would not be supported by the implementation, signals limits. The caller obtains a read lock on the result.

fill_copy = proc (lb, cnt, elem: t) returns (atomic_array[1])
signals (negative_size, limits, failure(string))
requires $t$ has copy; proctype $t$ returns $t$ signals failure(string)
effects The effect is like fill except that $elem$ is copied $cnt$ times. If $cnt < 0$, signals negative_size. Normally returns a new array with low bound $lb$ and size $cnt$ and with each element a copy of $elem$, as produced by $t$copy. The caller obtains a read lock on the result. Any failure signal raised by $t$copy is immediately re-signalled. This operation does not originate any failure signals by itself. If the new array cannot be represented by the implementation, signals limits.

add = proc (a: atomic_array[1], elem: t) signals (limits)
modifies a.
effects Obtains a write lock on $a$. If extending $a$ on the high end would cause the high bound or size of $a$ to be outside the range supported by the implementation, then signals limits. Otherwise extends $a$ by 1 in the high direction, and stores $elem$ as the new element. That is, $a_{\text{pos}}[\text{high}(a_{\text{pos}})+1] = elem$. 
add = proc (a: atomic_array[t], elem: t) signals (limits)
    modifies a.
    effects Obtains a write lock on a. If extending a on the low end would cause the low bound or size of a to be outside the range supported by the implementation, then signals limits. Otherwise extends a by 1 in the low direction, and stores elem as the new element. That is, \( a_{low}(a_{pre}) - 1 \) = elem.

remh = proc (a: atomic_array[t]) returns (t) signals (bounds)
    modifies a.
    effects Obtains a write lock on a. If a is empty, signals bounds. Otherwise shrinks a by removing its high element, and returns the removed element. That is, \( a_{low}(a_{pre}) = \text{high}(a_{pre}) - 1 \).

reml = proc (a: atomic_array[t]) returns (t) signals (bounds)
    modifies a.
    effects Obtains a write lock on a. If a is empty, signals bounds. Otherwise shrinks a by removing its low element, and returns the removed element. That is, \( \text{low}(a_{pre}) = \text{low}(a_{pre}) + 1 \).

set_low = proc (a: atomic_array[t], lb: int) signals (limits)
    modifies a.
    effects Obtains a write lock on a. If the new low (or high) bound would not be supported by the implementation, then signals limits. Otherwise, modifies the low and high bounds of a; the new low bound of a is lb and the new high bound is \( \text{high}(a_{pre}) = \text{low}(a_{pre}) + \text{lb} \).

trim = proc (a: atomic_array[t], lb, cnt: int) signals (negative_size, bounds)
    modifies a.
    effects If cnt < 0, signals negative_size and does not obtain any locks. Otherwise obtains a write lock on a. If \( lb < \text{low}(a) \) or \( lb > \text{high}(a) + 1 \), signals bounds. Otherwise, modifies a by removing all elements with index \( < lb \) or greater than or equal to \( lb + \text{cnt} \); the new low bound is \( lb \). For example, if \( a = \text{atomic_array}[\text{int}][1,2,3,4,5] \), then:
    trim(a, 2, 2) results in a having value \( \text{atomic_array}[\text{int}][2, 3] \)
    trim(a, 4, 3) results in a having value \( \text{atomic_array}[\text{int}][4, 5] \)

store = proc (a: atomic_array[t], i: int, elem: t) signals (bounds)
    modifies a.
    effects Obtains a write lock on a. If \( i < \text{low}(a) \) or \( i > \text{high}(a) \), signals bounds; otherwise makes elem the element of a with index i.

fetch = proc (a: atomic_array[t], i: int) returns (t) signals (bounds)
    effects If \( i < \text{low}(a) \) or \( i > \text{high}(a) \), signals bounds; otherwise returns the element of a with index i. Always obtains a read lock on a.

bottom = proc (a: atomic_array[t]) returns (t) signals (bounds)
    effects If a is empty, signals bounds; otherwise returns \( a[\text{low}(a)] \). Always obtains a read lock on a.

top = proc (a: atomic_array[t]) returns (t) signals (bounds)
    effects If a is empty, signals bounds; otherwise returns \( a[\text{high}(a)] \). Always obtains a read lock on a.

empty = proc (a: atomic_array[t]) returns (bool)
    effects Returns true if a contains no elements, returns false otherwise. In either case obtains a read lock on a.

dim = proc (a: atomic_array[t]) returns (int)
    effects Returns a count of the number of elements of a, obtains a read lock on a.
low = proc (a: atomic_array[][]) returns (int)
  effects Returns the low bound of a, obtains a read lock on a

high = proc (a: atomic_array[][]) returns (int)
  effects Returns the high bound of a, obtains a read lock on a

elements = iter (a: atomic_array[][]) yields (i) signals (failure(string))
  effects Obtains a read lock on a and yields the elements of a, each exactly once for each
  index, from the low bound to the high bound (i.e., bottom(a.pre), ..., top(a.pre)). The
  elements are fetched one at a time, using the indexes that were legal at the start of the
  call. If, during the iteration, a is modified so that fetching at a previously legal index
  signals bounds, then the iterator signals failure with the string "bounds". The iterator is
  divisible at yields.

indexes = iter (a: atomic_array[][]) yields (int)
  effects Obtains a read lock on a, then yields the indexes of a from the low bound of a.pre to
  the high bound of a.pre. Note that indexes is unaffected by any modifications done by the
  loop body. It is divisible at yields.

aa2a = proc (aa: atomic_array[][]) returns (array[])
  effects Obtains a read lock on aa and returns an array a with the same (sequential) state.

a2aa = proc (array[]) returns (aa: atomic_array[][])
  effects Returns an atomic_array aa with the same state as a. Obtains a read lock on aa.

equal = proc (a1, a2: atomic_array[][]) returns (bool)
  effects Returns true if a1 and a2 refer to the same atomic_array object; otherwise returns
  false. No locks are obtained.

similar = proc (a1, a2: atomic_array[][]) returns (bool) signals (failure(string))
  requires t has similar: proc type (t, t) returns (bool) signals (failure(string))
  effects Returns true if a1 and a2 have the same low and high bounds and if their elements
  are pairwise similar as determined by $similar. See the description of the similar
  operation of array for an equivalent body of code. This operation is divisible at calls to
  $similar. Read locks are obtained on a1 and a2, in that order.

similar1 = proc (a1, a2: atomic_array[][]) returns (bool) signals (failure(string))
  requires t has equal: proc type (t, t) returns (bool) signals (failure(string))
  effects Returns true if a1 and a2 have the same low and high bounds and if their elements
  are pairwise equal as determined by $equal. This operation works the same way as
  similar, except that $equal is used instead of $similar. Read locks are obtained on a1
  and a2, in that order.

copy = proc (a: atomic_array[][]) returns (b: atomic_array[][]) signals (failure(string))
  requires t has copy: proc type (t) returns () signals (failure(string))
  effects Returns a new atomic_array b with the same low and high bounds as a and such
  that each element b[i] contains $copy(a[i]). See the description of the copy operation of
  array for an equivalent body of code. This operation is divisible at calls to $copy, and
  obtains read locks on a and b.

copy1 = proc (a: atomic_array[][]) returns (b: atomic_array[][])
  effects Returns a new atomic_array b with the same low and high bounds as a and such
  that each element b[i] contains the same element as a[i]. Read locks are obtained on a
  and b.
transmit = proc (a: atomic_array[t]) returns (b: atomic_array[t]) signals (failure(string))
  requires f has transmit
  effects Returns a new array b with the same low and high bounds as a and such that each
  element b[i] contains a transmitted copy of a[i]. Read locks are obtained on a and b.
  Sharing among the elements of a is preserved in b. Signals failure if b cannot be
  represented on the receiving end or if fetching an element at a legal index of a_pre causes
  a bounds exception and ressignals any failure signals raised by f$transmit.

test_and_read = proc (aa: atomic_array[t]) returns (bool)
  effects Tries to obtain a read lock on aa. If the lock is obtained, returns true; otherwise no
  lock is obtained and the operation returns false. The operation does not "wait" for a lock.
  Even if the executing action "knows" that a lock could be obtained, false may be
  returned. Even if false is returned, a subsequent attempt to obtain a read lock might
  succeed without waiting.

test_and_write = proc (aa: atomic_array[t]) returns (bool)
  effects Tries to obtain a write lock on aa. If the lock is obtained, returns true; otherwise no
  lock is obtained and the operation returns false. The operation does not "wait" for a lock.
  Even if the executing action "knows" that a lock could be obtained, false may be
  returned. Even if false is returned, a subsequent attempt to obtain a write lock might
  succeed without waiting.

can_read = proc (aa: atomic_array[t]) returns (bool)
  effects Returns true if a read lock could be obtained on aa without waiting, otherwise
  returns false. No lock is actually obtained. Even if the executing action "knows" that a
  lock could be obtained, false may be returned. Since some concurrent action may obtain
  or release a lock on an atomic_array at any time, the information returned is unreliable:
  even if true is returned, a subsequent attempt to obtain the lock may require waiting; and
  even if false is returned, a subsequent attempt to obtain a read lock might succeed
  without waiting.

can_write = proc (aa: atomic_array[t]) returns (bool)
  effects Returns true if a write lock could be obtained on aa without waiting, otherwise
  returns false. No lock is actually obtained. Even if the executing action "knows" that a
  lock could be obtained, false may be returned. Since some concurrent action may obtain
  or release a lock on an atomic_array at any time, the information returned is unreliable:
  even if true is returned, a subsequent attempt to obtain the lock may require waiting; and
  even if false is returned, a subsequent attempt to obtain a write lock might succeed
  without waiting.

read_lock = proc (aa: atomic_array[t])
  effects Obtains a read lock on aa.

write_lock = proc (aa: atomic_array[t])
  effects Obtains a write lock on aa.
II.11. Structures

A struct (short for "structure") is an expandable collection of zero or more named objects. The
names are called members, and the types of its members may have different types.

An instantiation of struct has the form:

```plaintext
struct [ field_names, ... ]
```

where

```plaintext
field_names [ the name, ... ; type, optional ]
```

(see Appendix C). Like pointers, structures can be manipulated by value, by reference,
and by address. Like arrays, structures have a size and type:

```plaintext
struct_name size: size
```

A struct is compared by value if the components are comparable
objects.

For purposes of the math operations, an individual struct is manipulated as an ordered
tuple. In this context, the struct becomes a vector. The components of the vector
remain the same but are manipulated as a whole.

Much as with sequences, a struct is compared by element, not by members of objects; therefore, a
struct is comparable if all its components are comparable objects.

In the following operation descriptions, let `struct_name` be the struct.

Operations

```plaintext
replace_n: (x, y, z) => (x, z, y)
```

which replaces the component of a struct named `z` with the
component of a struct named `y`.

```plaintext
get_n: (n, m) => (n, m)
```

which returns the component of a struct named `m`.

```plaintext
set_x: (x, y, z) => (x, y, z)
```

which sets the component of a struct named `z` with the
component of a struct named `y`.

```plaintext
equal: (x, y, z) => (x, y, z)
```

which sets the component of a struct named `z` with the
component of a struct named `y`.

```plaintext
size: (x, y, z, m) => (x, y, z, m)
```

which sets the component of a struct named `m` with the
component of a struct named `y`.
similar = proc (s1, s2: st) returns (bool) signals (failure(string))
  requires each t_i has similar: proc type (t_i) returns (bool) signals (failure(string))
  effects Returns true if s1 and s2 contain similar objects for each component as determined
  by the t_i$\text{similar}$ operations. Any failure signal is immediately ressignalled. This operation
does not itself originate any failure signal. The comparison is done in lexicographic order
of the selectors; if any comparison returns false, false is returned immediately.

copy = proc (s: st) returns (st) signals (failure(string))
  requires each t_i has copy: proc type (t_i) returns (t_i) signals (failure(string))
  effects Returns a struct containing a copy of each component of s; copies are obtained by
  calling the t_i$\text{copy}$ operations. Any failure signal is immediately ressignalled. This operation
does not itself originate any failure signal. Copying is done in lexicographic order
of the selectors.

transmit = proc (s: st) returns (st) signals (failure(string))
  requires each t_i has transmit
  effects Returns a struct containing a transmitted copy of each component of s. Sharing is
  preserved among the components of s. Any failure signal from t_i$\text{transmit}$ is
  immediately ressignalled. This operation does not itself originate any failure signal.

II.12. Records

record = data type [n_1: t_1, ..., n_k: t_k] is r_gets_r, r_gets_s, set_n_1, ..., set_n_k, get_n_1, ..., get_n_k,
  equal, similar, similar1, copy, copy1, transmit

Overview

A record is a mutable collection of one or more named objects. The names are called selectors,
and the objects are called components. Different components may have different types. A record
also has an identity as an object.

An instantiation of record has the form:

record [field_spec, ...]

where

  field_spec ::= name, ..., type_actual

(see Appendix I). Selectors must be unique within an instantiation (ignoring capitalization), but the
ordering and grouping of selectors is unimportant. For example, the following name the same
type:

record[last, first, middle: string, age: int]
record[last: string, age: int, first, middle: string]

A record is created using a record constructor, see Section 6.2.11.

For purposes of the certain operations, the the names of the selectors are ordered
lexicographically. Lexicographic ordering of the selectors is the alphabetic ordering of the selector
names written in lower case (based on the ASCII ordering of characters).

In the following definitions of record operations, let \( r = \text{record}[n_1: t_1, ..., n_k: t_k] \).

Operations

  r_gets_r = proc (r1, r2: rt)
  modifies \( r1 \).
  effects Sets each component of \( r1 \) to be the corresponding component of \( r2 \).
r_gets_s = proc (r: r, s: st)
    modifies r.
    effects Have s is a struct type whose components have the same selectors and types as r.
    Sets each component of r to be the corresponding component of s.

set_n_i = proc (r: r, s: t)
    modifies r.
    effects Modifies r by making the component whose selector is n_i be s. There is a set_ operation for each selector.

got_n_i = proc (r: r) returns (t)
    effects Returns the component of r whose selector is n_i. There is a get_ operation for each selector.

equal = proc (r1, r2: r) returns (bool)
    effects Returns true if r1 and r2 are the same record object; otherwise returns false.

similar = proc (r1, r2: r) returns (bool)\!
    requires Each n_i has equal; procedure r is compatible with r.
    effects Returns true if r1 and r2 have the same record component as determined by the get_ and set_ operations. Any component that is not set is ignored. This operation does not itself execute any actions on the record objects in lexicographic order of the selectors; if any component causes error, it occurs immediately.

similar1 = proc (r1, r2: r) returns (bool)
    requires Each n_i has equal; procedure r is compatible with r.
    effects Returns true if r1 and r2 have the same record component as determined by the get_ and set_ operations. Any component that is not set is ignored. This operation does not itself execute any actions on the record objects in lexicographic order of the selectors; if any component causes error, it occurs immediately.

copy = proc (r: r) returns (r)
    requires Each n_i has copy; procedure r is compatible with r.
    effects Returns a new record containing a copy of each component of r. Copies are obtained by calling the get_ and set_ operations. Any component that is not set is ignored. This operation does not itself execute any actions on the record objects in lexicographic order of the selectors.

copy1 = proc (r: r) returns (r)
    effects Returns a new record containing the components of r as its components.

transmit = proc (r: r) returns (r)
    requires Each n_i has transmit
    effects Returns a new record containing a transmitted copy of each component of r.
    Shaping is preserved among the components of r. Any component that is not get_able is immediately recognized. This operation does not itself execute any actions on the record objects.
II.13. Atomic Records

atomic_record = data type \([n_1 : t_1, \ldots, n_k : t_k]\) is \(ar\text{\_}gets\_ar\), \(set\_n_1\), \(\ldots\), \(set\_n_k\), \(get\_n_1\), \(\ldots\), \(get\_n_k\),
\(ar2r\), \(r2ar\), \(equal\), \(similar\), \(similar1\), \(copy\), \(copy1\), \(transmit\),
\(test\_and\_read\), \(test\_and\_write\), \(can\_read\), \(can\_write\), \(read\_lock\), \(write\_lock\)

Overview

An atomic_record is a mutable atomic collection of one or more named objects. The names are
called selectors, and the objects are called components. Different components may have different
types. An atomic_record also has an identity as an object.

An instantiation of atomic_record has the form:

\[
\text{atomic\_record} [\text{field\_spec, \ldots}]
\]

where

\[
\text{field\_spec ::= name, \ldots : type\_spec}
\]

(see Appendix I). Selectors must be unique within an instantiation (ignoring capitalization), but the
ordering and grouping of selectors is unimportant. For example, the following name the same type:

\[
\begin{align*}
\text{atomic\_record}[\text{last, first, middle: string, age: int}] \\
\text{atomic\_record}[\text{last: string, age: int, first, middle: string}]
\end{align*}
\]

An atomic_record is created using a atomic_record constructor, see Section 6.2.11.

For purposes of the certain operations, the the names of the selectors are ordered
lexicographically. Lexicographic ordering of the selectors is the alphabetic ordering of the selector
names written in lower case (based on the ASCII ordering of characters).

Atomic_records use read/write locking to achieve atomicity. The locking rules are described in
Section 2.2.2. It is an error if a process that is not in an action attempts to test or obtain a lock;
when this happens the guardian running the process will crash. As defined below, the only
operation that (in the normal case) does not attempt to test or obtain a lock is the equal operation.

In the following, let \(ar = \text{atomic\_record}[n_1 : t_1, \ldots, n_k : t_k]\).

Operations

\(ar\text{\_}gets\_ar = \text{proc} (r_1, r_2 : ar)\)
\(\text{modify} r_1.\)
\(\text{effects} \) Obtains a write lock on \(r_1\) and a read lock on \(r_2\), then sets each component of \(r_1\) to
be the corresponding component of \(r_2\).

\(get\_n_i = \text{proc} (r : ar) \text{ returns} (t_i)\)
\(\text{effects} \) Obtains a read lock on \(r\) and returns the component of \(r\) whose selector is \(n_i\). There
is a \(get\) operation for each selector.

\(set\_n_i = \text{proc} (r : ar, e : t_i)\)
\(\text{modify} r.\)
\(\text{effects} \) Obtains a write lock on \(r\) and modifies \(r\) by making the component whose selector is
\(n_i\) be \(e\). There is a \(set\) operation for each selector.

\(ar2r = \text{proc} (ar : ar) \text{ returns} (r : ar)\)
\(\text{effects} \) Obtains a read lock on \(ar\) and returns a record \(r\) with the same state.

\(r2ar = \text{proc} (r : ar) \text{ returns} (ar : ar)\)
\(\text{effects} \) returns an atomic_record \(ar\) with the same state as \(r\). Obtains a read lock on \(ar\).
equal = proc (r1, r2: art) returns (bool)
effects Returns true if r1 and r2 are the very same atomic_record object; otherwise returns false. No locks are obtained.

similar = proc (r1, r2: art) returns (bool) signals (failure(string))
  requires each t_i has similar: proctype (t_i, t_j) returns (bool) signals (failure(string))
effects Obtains a read lock on r1, then a read lock on r2; then compares corresponding components from r1 and r2 using the similar operations. Any failure signal is immediately resignedalled. This operation does not itself originate any failure signal. The comparison is done in lexicographic order of the selectors; if any comparison returns false, false is returned immediately. If all comparisons return true, returns true.

similar1 = proc (r1, r2: art) returns (bool) signals (failure(string))
  requires each t_i has equal: proctype (t_i, t_j) returns (bool) signals (failure(string))
effects This operation is the same as similar, except that $equal$ is used instead of $similar$.

copy = proc (r: art) returns (res: art) signals (failure(string))
  requires each t_i has copy: proctype (t_i, t_j) returns (t_j) signals (failure(string))
effects Obtains a read lock on r, then returns a new atomic_record res obtained by performing $copy$(r) and then replacing each component with a copy of the corresponding component of r. Copies are obtained by calling the $copy$ operations. Any failure signal is immediately resignedalled. This operation does not itself originate any failure signal. Copying is done in lexicographic order of the selectors. A read lock is also obtained on the new atomic_record res.

copy1 = proc (r: art) returns (res: art)
effects Obtains a read lock on r, then returns a new atomic_record res containing the components of r as its components. A read lock is also obtained on the new atomic_record res.

transmit = proc (ar: art) returns (art) signals (failure(string))
  requires each t_i has transmit
  effects Returns a new atomic_record containing a transmitted copy of each component of ar. Sharing is preserved among the components of ar. A read lock is obtained on ar and the new atomic_array. Any failure signal from $transmit$ is immediately resignedalled. This operation does not itself originate any failure signal.

test_and_read = proc (ar: art) returns (bool)
effects Tries to obtain a read lock on ar. If the lock is obtained, returns true; otherwise no lock is obtained and the operation returns false. The operation does not "wait" for a lock. Even if the executing action "knows" that a lock could be obtained, false may be returned. Even if false is returned, a subsequent attempt to obtain a read lock might succeed without waiting.

test_and_write = proc (ar: art) returns (bool)
effects Tries to obtain a write lock on ar. If the lock is obtained, returns true; otherwise no lock is obtained and the operation returns false. The operation does not "wait" for a lock. Even if the executing action "knows" that a lock could be obtained, false may be returned. Even if false is returned, a subsequent attempt to obtain a write lock might succeed without waiting.
can_read = proc (ar:art) returns (bool)
effects Returns true if a read lock could be obtained on ar without waiting, otherwise returns false. No lock is actually obtained. If the assuming action "knows" that a lock could be obtained, false may be returned. Otherwise, a concurrent action may obtain or release a lock on an atomic record, or an atomic record may require waiting; and even if true is returned, a subsequent attempt to obtain a read lock might succeed without waiting.

can_write = proc (ar:art) returns (bool)
effects Returns true if a write lock could be obtained on ar without waiting, otherwise returns false. No lock is actually obtained. If the assuming action "knows" that a lock could be obtained, false may be returned. Otherwise, a concurrent action may obtain or release a lock on an atomic record, or an atomic record may require waiting; and even if true is returned, a subsequent attempt to obtain a write lock might succeed without waiting.

read_lock = proc (ar:art)
effects Obtains a read lock on ar.

write_lock = proc (ar:art)
effects Obtains a write lock on ar.

II.14. Oneofs

oneof = data type [(n_1: t_1, ..., n_k: t_k) is make_n_1, ..., make_n_k, is_n_1, ..., is_n_k, value_n_1, ..., value_n_k, cdiv, vdiv, equal, similar, copy, transmit]

Overview

A oneof is a tagged, discriminated union; that is, a labeled object, to be thought of as "one of" a set of alternatives. The label is called the tagpart, and the object is called the value or data part.

An instantiation of oneof has the form:

    oneof [field_spec, ...]

where (as for records)

    field_spec := name, ..., : type_actual

(see Appendix G). Tags must be unique within an instantiation (ignoring capitalization), but the ordering and grouping of tags is unimportant.

Although there are oneof operations for decomposing oneof objects, they are usually decomposed via the taggeta statement, which is discussed in Section 10.14.

A oneof is immutable but may contain a mutable object; therefore, a oneof is atomic only if all of the types of its data parts are atomic.

In the following, let ot = oneof[n_1: t_1, ..., n_k: t_k].

Operations

make_n_1 = proc (e: t_1) returns (ot)
effects Returns a oneof object with tag n_1 and value e. There is a make_... operation for each selector.

is_n_1 = proc (o: ot) returns (bool)
effects Returns true if the tag of o is n_1, else returns false. There is an is_... operation for each selector.
value_n_i = proc (o: ot) returns (t_j) signals (wrong_tag)
effects If the tag of o is n_i, returns the value of o; otherwise signals wrong_tag. There is a
type_operation for each selector.

c2v = proc (o: ot) returns (vt)
effects Here vt is a variant type with the same selectors and types as ot. Returns a new
variant object with the same tag and value as o.

v2o = proc (v: vt) returns (ot)
effects Here vt is a variant type with the same selectors and types as ot. Returns a oneof
object with the same tag and value as v.

equal = proc (o1, o2: ot) returns (bool) signals (failure(string))
requires each t_j has equal: proctype (t_j, t_j) returns (bool) signals (failure(string))
effects Returns true if o1 and o2 have the same tag and equal values as determined by the
equal operation of their data part's type. Any failure signal is immediately resignalled.
This operation does not itself originate any failure signal. This operation is divisible at the
call of t_jequal.

similar = proc (o1, o2: ot) returns (bool) signals (failure(string))
requires each t_j has similar: proctype (t_j, t_j) returns (bool) signals (failure(string))
effects Returns true if o1 and o2 have the same tag and similar values as determined by the
similar operation of their value's type. Any failure signal is immediately resignalled.
This operation does not itself originate any failure signal. This operation is divisible at the
call of t_jsimilar.

copy = proc (o: ot) returns (ot) signals (failure(string))
requires each t_j has copy: proctype (t_j) returns (t_j) signals (failure(string))
effects Returns a oneof object with the same tag as o and containing a value a copy of
o's value; the copy is made during the copy operation of the value's type. Any failure
signal is immediately resignalled. This operation does not itself originate any failure
signal. This operation is divisible at the call of t_jcopy.

transmit = proc (o: ot) returns (ot) signals (failure(string))
requires each t_j has transmit
effects Returns a oneof object with the same tag as o and containing a value a
transmitted copy of o's value. Any failure signal is immediately resignalled. This
operation does not itself originate any failure signal.

II.15. Variants

variant = data type [n_1, t_1, ..., n_k, t_k] is make_n_1, ..., make_n_k, change_n_1, ..., change_n_k,
is_n_1, ..., is_n_k, value_n_1, ..., value_n_k, v_gets_v, v_gets_o,
equal, similar, similar1, copy, copy1, transmit

Overview

A variant is a mutable, tagged, discriminated union. Its state is a oneof, that is, a labeled object,
to be thought of as "one of" a set of alternatives. The label is called the tag part, and the object is
called the value (or data part). A variant also has an identity as an object.

An instantiation of variant has the form:

    variant [ field_spec , ... ]

where

field_spec ::= name, ..., : type_actual
(see Appendix I). Tags must be unique within an instantiation (ignoring capitalization), but the
ordering and grouping of tags is unimportant.
Although there are variant operations for decomposing variant objects, they are usually decomposed via the \texttt{tagcase} statement, which is discussed in Section 10.14.

In the following let \( \text{vt} = \text{variant}(n_1: t_1, \ldots, n_k: t_k) \).

\textbf{Operations}

\texttt{make	extunderscore n} = proc \((e: t)\) returns \((\text{vt})\)
  \hspace{1em} effects Returns a new variant object with tag \(n_i\) and value \(e\). There is a \texttt{make	extunderscore} operation for each selector.

\texttt{change	extunderscore n} = proc \((v: \text{vt}, e: t)\)
  \hspace{1em} modifies \(v\).
  \hspace{1em} effects Modifies \(v\) to have tag \(n_i\) and value \(e\). There is a \texttt{change	extunderscore} operation for each selector.

\texttt{is	extunderscore n} = proc \((v: \text{vt})\) returns \((\text{bool})\)
  \hspace{1em} effects Returns true if the tag of \(v\) is \(n_i\); otherwise returns false. There is an \texttt{is	extunderscore} operation for each selector.

\texttt{value	extunderscore n} = proc \((v: \text{vt})\) returns \((t)\) signals \((\text{wrong	extunderscore tag})\)
  \hspace{1em} effects If the tag of \(v\) is \(n_i\), returns the value of \(v\); otherwise signals \texttt{wrong	extunderscore tag}. There is a \texttt{value	extunderscore} operation for each selector.

\texttt{v	extunderscore gets\textunderscore v} = proc \((v1, v2: \text{vt})\)
  \hspace{1em} modifies \(v1\).
  \hspace{1em} effects Modifies \(v1\) to contain the same tag and value as \(v2\).

\texttt{v	extunderscore gets\textunderscore o} = proc \((v: \text{vt}, o: ot)\)
  \hspace{1em} modifies \(v\).
  \hspace{1em} effects Here \(ot\) is the oneof type with the same selectors and types as \(vt\). Modifies \(v\) to contain the same tag and value as \(o\).

\texttt{equal} = proc \((v1, v2: \text{vt})\) returns \((\text{bool})\)
  \hspace{1em} effects Returns true if \(v1\) and \(v2\) are the same variant object.

\texttt{similar} = proc \((v1, v2: \text{vt})\) returns \((\text{bool})\) signals \((\text{failure(string)})\)
  \hspace{1em} requires each \(t\) has similar: proc\texttt{type} \((t, t)\) returns \((\text{bool})\) signals \((\text{failure(string)})\)
  \hspace{1em} effects Returns true if \(v1\) and \(v2\) have the same tag and similar values as determined by the \texttt{similar} operation of their value's type. Any failure signal is immediately resigalled. This operation does not itself originate any failure signal. This operation is divisible at the call of \texttt{t$\text{\$}similar}\).

\texttt{similar1} = proc \((v1, v2: \text{vt})\) returns \((\text{bool})\) signals \((\text{failure(string)})\)
  \hspace{1em} requires each \(t\) has equal: proc\texttt{type} \((t, t)\) returns \((\text{bool})\) signals \((\text{failure(string)})\)
  \hspace{1em} effects Same as similar, except that \(t$\text{\$equal}\) is used instead of \(t$\text{\$similar}\).

\texttt{copy} = proc \((v: \text{vt})\) returns \((\text{vt})\) signals \((\text{failure(string)})\)
  \hspace{1em} requires each \(t\) has copy: proc\texttt{type} \((t, t)\) returns \((\text{bool})\) signals \((\text{failure(string)})\)
  \hspace{1em} effects Returns a variant object with the same tag as \(v\) and containing as a value a copy of \(v\)'s value; the copy is made using the \texttt{copy} operation of the value's type. Any failure signal is immediately resigalled. This operation does not itself originate any failure signal. This operation is divisible at the call of \texttt{t$\text{\$copy} \)}.

\texttt{copy1} = proc \((v: \text{vt})\) returns \((\text{vt})\)
  \hspace{1em} effects Returns a new variant object with the same tag as \(v\) and containing \(v\)'s value as its value.
transmit = proc (v: vt) returns (vt) signals (failure(string))
   requires each t has transmit
   effects Returns a variant object with the same tag as v and containing as a value a
   transmitted copy of v's value. Any failure signal is immediately resignalled. This
   operation does not itself originate any failure signal.

II.16. Atomic Variants

atomic_variant = data type [n_1; t_1, ..., n_k; t_k] is make_n_1, ..., make_n_k, change_n_1, ..., change_n_k,
   av_gets_av, is_n_1, ..., is_n_k, value_n_1, ..., value_n_k, av2v, v2av,
   equal, similar, similar1, copy, copy1, transmit,
   test_and_read, test_and_write, can_read, can_write, read_lock, write_lock

Overview

An atomic_variant is a mutable, atomic, tagged, discriminated union. Its state is a oneof, that is, a
labeled object, to be thought of as "one of" a set of alternatives. The label is called the tag part,
and the object is called the value (or data part). An atomic_variant also has an identity as an
object.

An instantiation of atomic_variant has the form:
   atomic_variant [ field_spec , ... ]
where
   field_spec ::= name, ..., : type_actual
(see Appendix I). Tags must be unique within an instantiation (ignoring capitalization), but the
ordering and grouping of tags is unimportant.

Although there are atomic_variant operations for decomposing atomic_variant objects, they are
usually decomposed via the tagtest statement or tagwait statement, which are discussed in
Section 10.15.

In the following, let avt = atomic_variant[n_1: t_1, ..., n_k: t_k].

Operations

make_n_i = proc (e: t_i) returns (av: avt)
   effects Returns a new atomic_variant object av with tag n_i and value e. Obtains a read lock
   on av. There is a make_operation for each selector.

change_n_i = proc (v: avt, e: t_i)
   modifies v.
   effects Obtains a write lock on v, then modifies v to have tag n_i and value e. There is a
   change_operation for each selector.

avGets_av = proc (v1: v2: avt)
   modifies v1.
   effects Obtains a read lock on v2 and then a write lock on v1, then modifies v1 to contain
   the same tag and value as v2.

is_n_i = proc (v: avt) returns (bool)
   effects Obtains a read lock on v, then returns true if the tag of v is n_i; otherwise returns
   false. There is an is_operation for each selector.

value_n_i = proc (v: avt) returns (t_i) signals (wrong_tag)
   effects Obtains a read lock on v. Then, if the tag of v is n_i, returns the value of v; otherwise
   signals wrong_tag. There is a value_operation for each selector.
av2v = proc (av: avt) returns (v: vt)
  effects Here vt is a variant type with the same selectors and types as avt. Obtains a read
  lock on av and returns a variant v with the same state.

v2av = proc (v: vt) returns (av: avt)
  effects Here vt is a variant type with the same selectors and types as avt. Returns an
  atomic_variant av with the same state as v. Obtains a read lock on av.

equal = proc (v1, v2: avt) returns (bool)
  effects Returns true if v1 and v2 are the same atomic_variant object. No locks are
  obtained.

similar = proc (v1, v2: avt) returns (bool) signals (failure(string))
  requires each i has similar: proctype (ti, tj) returns (bool) signals (failure(string))
  effects Obtains read locks on v1 and v2, in order, and then compares the objects; returns
  true if v1 and v2 have the same tag and similar values as determined by the similar
  operation of their type. Any failure signal is immediately resignalled. This operation does
  not itself originate any failure signal. This operation is divisible at the call of t\$similar.

similar1 = proc (v1, v2: avt) returns (bool) signals (failure(string))
  requires each i has equal: proctype (ti, tj) returns (bool) signals (failure(string))
  effects Same as similar, except that t\$equal is used instead of t\$similar.

copy = proc (v: avt) returns (avt) signals (failure(string))
  requires each i has copy: proctype (ti) returns (tj) signals (failure(string))
  effects Obtains a read lock on v, then returns an atomic_variant object with the same tag as
  v and containing as a value a copy of v's value; the copy is made using the copy
  operation of the value's type. Any failure signal is immediately resignalled. This
  operation does not itself originate any failure signal. This operation is divisible at the call
  of t\$copy. A read lock is obtained on the result.

copy1 = proc (v: avt) returns (avt)
  effects Obtains a read lock on v, then returns a new atomic_variant object with the same tag
  as v and containing v's value as its value. A read lock is obtained on the result.

transmit = proc (v: avt) returns (avt) signals (failure(string))
  requires each i has transmit
  effects Returns an atomic_variant object with the same tag as v and containing as a value a
  transmitted copy of v's value. Obtains a read lock on v. Any failure signal is immediately
  resignalled. This operation does not itself originate any failure signal.

test_and_read = proc (av: avt) returns (bool)
  effects Tries to obtain a read lock on av. If the lock is obtained, returns true; otherwise no
  lock is obtained and the operation returns false. The operation does not "wait" for a lock.
  Even if the executing action "knows" that a lock could be obtained, false may be
  returned. Even if false is returned, a subsequent attempt to obtain a read lock might
  succeed without waiting.

test_and_write = proc (av: avt) returns (bool)
  effects Tries to obtain a write lock on av. If the lock is obtained, returns true; otherwise no
  lock is obtained and the operation returns false. The operation does not "wait" for a lock.
  Even if the executing action "knows" that a lock could be obtained, false may be
  returned. Even if false is returned, a subsequent attempt to obtain a write lock might
  succeed without waiting.
can_read = proc (av: avt) returns (bool)
effects Returns true if a read lock could be obtained on \textit{av} without waiting, otherwise returns false. No lock is actually obtained. Even if the executing action "knows" that a lock could be obtained, false may be returned. Since some concurrent action may obtain or release a lock on an \texttt{atomic\_variant} at any time, the information returned is unreliable: even if true is returned, a subsequent attempt to obtain the lock may require waiting; and even if false is returned, a subsequent attempt to obtain a read lock might succeed without waiting.

can_write = proc (av: avt) returns (bool)
effects Returns true if a write lock could be obtained on \textit{av} without waiting, otherwise returns false. No lock is actually obtained. Even if the executing action "knows" that a lock could be obtained, false may be returned. Since some concurrent action may obtain or release a lock on an \texttt{atomic\_variant} at any time, the information returned is unreliable: even if true is returned, a subsequent attempt to obtain the lock may require waiting; and even if false is returned, a subsequent attempt to obtain a write lock might succeed without waiting.

read_lock = proc (av: avt)
effects Obtains a read lock on \textit{av}.

write_lock = proc (av: avt)
effects Obtains a write lock on \textit{av}.

II.17. Procedures and Iterators
proctype = data type is equal, similar, copy
itertype = data type is equal, similar, copy

Overview

Procedures and iterators are objects created by the Argus system. The type specification for a procedure or iterator contains most of the information stated in a procedure or iterator heading; a procedure type specification has the form:

\texttt{proctype \{ \texttt{\{ \texttt{type\_spec, ...} \}} \} \{ \texttt{\{ \texttt{returns, ...} \}} \}\{ \texttt{\{ \texttt{signals} \}} \}

and an iterator type specification has the form:

\texttt{itertype \{ \texttt{\{ \texttt{type\_spec, ...} \}} \} \{ \texttt{\{ \texttt{yields, ...} \}} \}\{ \texttt{\{ \texttt{signals} \}} \}

where

\begin{align*}
\texttt{returns} & \quad \texttt{:= \; returns \{ \texttt{\{ \texttt{type\_spec, ...} \}} \}} \\
\texttt{yields} & \quad \texttt{:= \; yields \{ \texttt{\{ \texttt{type\_spec, ...} \}} \}} \\
\texttt{signals} & \quad \texttt{:= \; signals \{ \texttt{\{ \texttt{exception, ...} \}} \}} \\
\texttt{exception} & \quad \texttt{:= \; name \{ \texttt{\{ \texttt{type\_spec, ...} \}} \}}
\end{align*}

(see Appendix I). The first list of type specifications describes the number, types, and order of arguments. The \texttt{returns} or \texttt{yields} clause gives the number, types, and order of the objects to be returned or yielded. The \texttt{signals} clause lists the exceptions raised by the procedure or iterator; for each exception name, the number, types, and order of the objects to be returned are also given. All names used in a \texttt{signals} clause must be unique. The ordering of exceptions is not important. For example, both of the following type specifications name the procedure type for string\&sub:

\begin{align*}
\texttt{proctype \{ \texttt{string, int, int} \} \texttt{returns \{ \texttt{string} \} \texttt{signals \{ \texttt{bounds, negative\_size} \}}} \\
\texttt{proctype \{ \texttt{string, int, int} \} \texttt{returns \{ \texttt{string} \} \texttt{signals \{ \texttt{negative\_size, bounds} \}}}
\end{align*}
Procedure and Iterator objects are created by compiling modules (and by the bind expression, see Section 9.8). Procedure and iterator types are not transmissible and are considered to be immutable and atomic in normal use. However, some uses of own data (see Section 12.7) in procedures and iterators can violate this assumption.

In the following operation descriptions, \( t \) stands for a proctype or ltertype.

**Operations**

\[
\text{equal} = \text{proc} \ (x, y: t) \ \text{returns} \ (\text{bool}) \\
\text{similar} = \text{proc} \ (x, y: t) \ \text{returns} \ (\text{bool}) \\
\text{effects} \text{ These operations return true if and only if } x \text{ and } y \text{ are the same implementation of the same abstraction, with the same parameters (see Section 12.6).}
\]

\[
\text{copy} = \text{proc} \ (x: t) \ \text{returns} \ (t) \\
\text{effects} \text{ Returns } x.
\]

**II.18. Handlers and Creators**

\[
\text{handlertype} = \text{data type is equal, similar, copy, transmit} \\
\text{creatorotype} = \text{data type is equal, similar, copy, transmit}
\]

**Overview**

Handlers and creators are created by the Argus system. The type specification for a handler or creator contains most of the information stated in a handler or creator heading; a handler type specification has the form:

\[
\text{handlertype} \ (\text{[type spec , ... ]}) [\text{returns}] [\text{signals}]
\]
and a creator type specification has the form:

\[
\text{creatorotype} \ (\text{[type spec , ... ]}) [\text{returns}] [\text{signals}]
\]

where

\[
\text{returns} ::== \text{returns} (\text{type spec , ...}) \\
\text{signals} ::== \text{signals} (\text{exception , ...}) \\
\text{exception} ::== \text{name} [\text{(type spec , ...)}]
\]

(see Appendix I). The first list of type specifications describes the number, types, and order of arguments. The returns clause gives the number, types, and order of the objects to be returned. The signals clause lists the exceptions raised by the handler or creator; for each exception name, the number, types, and order of the objects to be returned are also given. All names used in a signals clause must be unique; none can be unavailable or failure, which have a pre-defined meaning for remote calls (see Section 8.3). The ordering of exceptions is not important.

Creators are created by compiling modules, and handlers are created as a side-effect of guardian creation. Handlers and creators are transmissible and are considered to be immutable and atomic in normal use. Certain uses of own data in handlers can violate this assumption.

In the following operation descriptions, \( t \) stands for a handlertype or creatorotype.

**Operations**

\[
\text{equal} = \text{proc} \ (x, y: t) \ \text{returns} \ (\text{bool}) \\
\text{similar} = \text{proc} \ (x, y: t) \ \text{returns} \ (\text{bool}) \\
\text{effects} \text{ These operations return true if and only if } x \text{ and } y \text{ are the same object (see Section 12.6 for an exact definition for the case of creators in guardian generators).}
\]
copy = proc (x: t) returns (t)
transmit = proc (x: t) returns (t)
effects Returns x.

II.19. Anys

any = data type is create, force, is_type

Overview
An object of type any contains a type T and an object of type T. Anys are immutable and are not transmissible. Anys are atomic only if their contained object is atomic.

Operations
create = proc[T: type] (contents: T) returns (any)
effects Returns an any object containing contents and the type T.

force = proc[T: type] (thing: any) returns (T) signals (wrong_type)
effects If thing contains an object of a type included in type T, then that object is returned; otherwise wrong_type is signalled.

is_type = proc[T: type] (thing: any) returns (bool)
effects If thing contains an object of a type included in type T, then true is returned; otherwise, false is returned.

II.20. Images

image = data type is create, force, is_type, copy, transmit

Overview
An object of type image is the value of an arbitrary transmissible type. See Section 14 for more details. Images are immutable, atomic, and transmissible.

Operations
create = proc[T: type] (contents: T) returns (image) signals (failure,string)
requires T has transmit
effects Returns an image object obtained from contents via the encode operation of T. Resignals any failure signal raised by T's encode operation.

force = proc[T: type] (thing: image) returns (T) signals (wrong_type, failure,string)
requires T has transmit
effects If thing encodes an object of a type included in type T, then that object is extracted using the decode operation of T and returned. Otherwise wrong_type is signalled. Resignals any failure signal raised by T's decode operation.

is_type = proc[T: type] (thing: image) returns (bool)
requires T has transmit
effects If thing encodes an object of a type included in type T, then true is returned; otherwise, false is returned.

copy = proc (thing: image) returns (image)
transmit = proc (thing: image) returns (image)
effects Returns thing.
II.21 Mutexes

mutex = data type[t: type] is create, set_value, get_value, changed, equal, similar, copy, transmit

Overview

A mutex is a mutable container for an object of type t. A mutex also has an identity as an object.

An object of type mutex[t] provides mutual exclusion for process synchronization, and allows explicit control over how information contained in the mutex is written to stable storage (see Section 15.1).

The seize statement is used in order to gain possession of a mutex. See section 6.7.

Although mutex objects are mutable, sharing among mutex objects is usually wrong, because the contained object should only be accessible through the mutex. Hence there is no copy operation, since this would introduce sharing, and there is no similar operation to check for sharing (see Section 6.7).

Operations

create = proc (thing: t) returns (mutex[t])
  effects Returns a new mutex object containing thing.

set_value = proc (container: mutex[t], contents: t)
  modifies container.
  effects Modifies container by replacing its contained object with contents.

get_value = proc (container: mutex[t]) returns (t)
  effects Returns the object contained in container.

changed = proc (container: mutex[t])
  effects Informs the Argus system that the calling action requires the contents of container to be copied to stable storage by the time the action commits, provided container is accessible from a stable variable. It is a programming error if a process that is not running an action calls this operations, and if this is done the guardian will crash.

equal = proc (m1, m2: mutex[t]) returns (bool)
  effects Returns true if and only if m1 and m2 are the same object.

similar = proc (m1, m2: mutex[t]) returns (bool) signals (failure(string))
  requires t has similar: proctype(t, t) returns(bool) signals (failure(string))
  effects Seizes m1, then seizes m2, and calls $similar to determine its result; any failure signal is immediately resignalled. Possession of both mutexes is retained until $similar terminates.

copy = proc (m1: mutex[t]) returns (m2: mutex[t]) signals (failure(string))
  requires t has copy: proctype(t, t) returns(bool) signals (failure(string))
  effects Seizes m1, then calls $copy to make a copy which it places in the new mutex object m2. Any failure signal is immediately resignalled. Possession of m1 is retained until $copy terminates.

transmit = proc (m1: mutex[t]) returns (mutex[t]) signals (failure(string))
  requires t has transmit
  effects Seizes m1, and returns a new mutex containing a transmitted copy of the contained object. Any failure signal is immediately resignalled. Possession of m1 is retained until $transmit terminates.
Appendix III
Rules and Guidelines for Using Argus

This appendix collects the rules and guidelines that should be followed when programming in Argus. Following these rules makes `setze` statements meaningful, actions atomic, and so on. In some rare cases there may be valid reasons for violating these guidelines, but doing so greatly increases the difficulty of building, debugging, and running the resulting system.

All of the rules listed in this appendix are based on information appearing elsewhere in the manual. Each rule is followed by a brief rationale, including a reference to the section of the manual from which it is drawn.

III.1. Serializability and Actions

* Actions should share only atomic objects.

Rationale: Actions that share non-atomic data are not necessarily serializable. [Section 2.2.2]

* A subaction that aborts should not return any information obtained from data shared with other concurrent actions.

Rationale: Returning such data may violate serializability. [Section 2.2.1]

* A nested topaction should be serializable before its parent. This is true if either
  1. the nested topaction performs a benevolent side effect (a change to the state of the representation that does not affect the abstract state), or
  2. all communication between the nested topaction and its parent is through atomic objects.

Rationale: Other uses may violate serializability. [Section 2.2.3]

* The creation or destruction of a guardian must be synchronized with the use of that guardian via atomic objects such as the catalog.

Rationale: Otherwise serializability may be violated. [Section 10.18]

III.2. Actions and Exceptions

* If an exception raised by a call should not commit an action, the exception must be handled within that action.

Rationale: If an exception raised within an action body is handled outside the action, the implicit flow of control outside of the action will commit the action. [Section 11.5]
III.3. Stable Variables

- Stable variables should denote resilient data objects.

  *Rationale:* Only data objects that are (reachable from the stable variables and) resilient are written to stable storage when a topadition commits. (This can be ensured by having stable variables only denote objects of an atomic type or objects protected by mutex.) Non-resilient objects stored in stable variables are only written to stable storage when the guardian is created. [Section 13.1]

- If a bound procedure or iterator will be accessible from a stable variable,
  1. the procedure or iterator being bound must be atomic and
  2. only atomic objects should be bound as arguments.

  *Rationale:* The bound procedure or iterator may be stored in stable storage, and non-atomic data is only written to stable storage once. [Section 9.8]

III.4. Transmission and Transmissibility

- An abstract type’s encode and decode operations should not cause side effects.

  *Rationale:* The number of calls to an encode or decode operation is unpredictable, since arguments or results may be encoded and decoded several times as the system tries to establish communication. In addition, verifying the correctness of transmission is easier if encode and decode are simply transformations to and from the external representation. [Section 14.3]

- If the naming relation among objects to be transmitted is cyclic (e.g., a circular list) then encode and decode must be implemented in one of two ways:
  1. The internal and external representation types must be identical, and encode and decode return their argument without modifying or accessing it, or
  2. The external representation object must be acyclic.

  *Rationale:* A circular external representation may cause decode to fail. [Section 14.4]

- Objects that share other objects should be bound into a handler or creator in the same bind expression.

  *Rationale:* Sharing is only preserved among objects bound at the same time. [Section 9.8]

III.5. Mutex

- Mutual exclusion or atomic data should be used to synchronize access to all shared objects.

  *Rationale:* In the presence of concurrency, any interleaving of indivisible events is possible. Without synchronization mechanisms, this concurrency will be visible to programs, significantly complicating coding and testing. [Section 8]
III.5 Mutex

- All modifications to mutex objects should be made inside `seize` statements.

  **Rationale:** The system will gain possession of a mutex object before writing it to stable storage; thus, seizing a mutex in order to modify it will prevent the system from copying a mutex object when it is in an inconsistent state. This also prevents other processes from seeing inconsistent data [Section 15.2 and Section 15.1]

- Nested seizes should be avoided when `pause` is used, and `pause` must be avoided when nested seizes are used.

  **Rationale:** A `pause` in a nested `seize` does not actually release possession of the mutex object. [Section 10.17]

- If an object is referred to by a mutex object, it should not be referred to by any other object, nor should it be denoted by a variable except when in possession of the containing mutex.

  **Rationale:** If an object contained in a mutex can be reached by a method other than seizing the mutex, the mutual exclusion property of the mutex is undermined. [Section 6.7]

- No activity that is likely to take a long time should be performed while in a `seize` statement. In particular, programs should not make handler calls or wait for locks on atomic objects while in possession of a mutex.

  **Rationale:** Waiting for a lock while in a mutex is likely to cause a deadlock with other actions or between the action holding the mutex and the Argus system. [Section 15.3]

- Mutex objects should not share data with one another, unless the shared data is atomic or mutex.

  **Rationale:** Sharing of non-atomic objects between mutex objects is not preserved when the mutexes are written to stable storage. [Section 15.3]

- Mutex[if]$changed must be called after the last modification (on behalf of some action) to the contained object of a mutex.

  **Rationale:** The Argus system is free to copy the mutex to stable storage as soon as mutex[if]$changed has been called. Changes after the last call to mutex[if]$changed but before topaction commit may not be written to stable storage. [Section 15.3]

- Mutex[if]$changed should be called even if the mutex object changed is not accessible from the stable variables.

  **Rationale:** In a scenario where the object was accessible, becomes inaccessible, then becomes accessible again, it is possible that stable storage would not be updated properly if this rule were not followed. The system guarantees that no problems with updating stable storage will arise if mutex[if]$changed is always called after the last modification to the object. [Section 15.3]
• An atomic type implemented with a representation consisting of several mutex objects should use separate topactions to ensure that the mutexes are written to stable storage in an order that preserves the correctness of the representation.

Rationale: Mutexes are written to stable storage incrementally. Sometimes, subtle timing problems can be caused by incremental writing if this rule is not followed. [Section 15.3]

III.6. User-Defined Atomic Objects

• If an atomic object X of type T provides operations \( O_1 \) and \( O_2 \), and action A has executed \( O_1 \) but not yet committed, then operation \( O_2 \) can be performed by a concurrent action B only if \( O_1 \) and \( O_2 \) commute: given the current state of X, the effect (as described by the sequential specification of T) of performing \( O_1 \), then \( O_2 \) is the same as performing \( O_2 \), then \( O_1 \). “Effect” includes both results returned and the (abstract) state modified.

Rationale: There are two concurrency constraints for user-defined atomic objects:
1. An action can observe the effects of other actions only if those actions committed relative to the first action.
2. Operations executed by one action cannot invalidate the results of operations executed by a concurrent action.

Two operations (or sequences of operations) that commute in their effect on the abstract state of X may be permitted to run concurrently, even if they do not commute in their effect on the representation of X. This distinction between an abstraction and its implementation is crucial in achieving reasonable performance. [Section 15.4]

• If a user-defined atomic object is accessible from the stable variables of some guardian, it should be written to stable storage whenever an action that modifies it commits to the top.

Rationale: A user-defined atomic type that is not written to stable storage on topaction commit will not be resilient. [Section 15.2]

• The form of the rep for a user-defined atomic type should be one of the following possibilities.
  1. The rep is itself atomic. Note that mutex is not an atomic type.
  2. The rep is mutex[t] where t is a synchronous type. For example, t could be atomic, or it could be the representation of an atomic type, if the operations on the this fictitious atomic type are coded in-line so that the entire type behaves atomically.
  3. The rep is an atomic collection of mutex types containing synchronous types.
  4. The rep is a mutable collection of synchronous types, and objects of the representation type are never modified after they are initialized. That is, mutation may be used to create the initial state of such an object, but once this has been done the object must never be modified.

Rationale: In any other case it will be impossible to guarantee the resilience or serializability of the type’s objects independently of how they are used. [Section 15.3]
III.7. Subordinate Where Clauses

- A where clause requirement on a cluster as a whole should be used whenever the actual parameters make some difference in the abstraction. For example, in a set cluster, the type parameter's equal operation must be required by the cluster as a whole, in order to preserve type safety and the representation invariant.

  *Rationale:* Argus assumes that requirements that are not placed on the cluster as a whole do not affect the semantics of the abstraction or the representation. [Section 12.6]
Appendix IV
Changes from CLU

This appendix lists the changes made to Argus that are not upward compatible with CLU, that is, those which are not merely additions to CLU and that would cause a CLU program to be illegal or to run differently.

IV.1. Exception Handling

Unlike CLU, which propagated unhandled exceptions (by turning them into failure exceptions) and gave the failure exception special status, unhandled exceptions in Argus are considered errors and always cause a crash of the guardian, and failure is not given special status. All exceptions signalled in a procedure, iterator, handler, or creator must be declared in the routine’s header, and there are no implicit resignals of failure exceptions. See Section 11.6 for details.

IV.2. Type Any

The type any is now a type like any other type, with parameterized routines force, create, and is_type. Thus the CLU manual’s notion of "type inclusion" is no longer necessary (but there is a new notion of type inclusion in Argus, see Section 6.1). The any$force routine only signals "wrong_type" if the any object’s underlying type is not included in the type parameter given, but the type of the result of any$force is its type parameter. The any$is_type routine returns false if the any object’s underlying type is not included in the type parameter given. The CLU reserved word "force" was eliminated from Argus, and the creation of an any object is never implicit in an assignment in Argus.

IV.3. Built-in Types

Several changes to the interfaces of the built-in types were necessitated by the changes to exception handling. Specifically, the following changes were made to the built-in types.

1. The string operations concat, append, s2ac, ac2s, a2sc, and ac2s, can now all signal limits. A string literal that would be too large to represent will not be compiled.

2. The sequence operations fill, fill_copy, addh, addt, and concat can now all signal limits. A sequence constructor that would be too large to represent will not be compiled.

3. The array (and atomic_array) operations create, predict, set_low, fill, fill_copy, addh, and addt can now all signal limits. An array constructor that cannot be legally represented will either not be compiled (if this can be detected at compile time) or will signal limits.

4. The copy operations of the structured built-in type generators, and the fill_copy operations of sequence and array (and atomic_array), allow the copy operations of their type parameters to have a failure(string) exception. They will resignal such a failure exception. (Note that the type inclusion rule allows a type parameter to be used even if its copy operation does not have exceptions.)

5. The similar operations of the built-in structured type generators allow the similar operations of their type parameters to have a failure(string) exception. They will resignal such a failure exception.

6. The equal operations of the type generators sequence, struct, and oneof, and the similart
operations of the type generators array, record, and variant (and their atomic counterparts), allow the equal operation of their type parameters to have a failure(string) exception. They will resignal such a failure exception.

7. The elements iterator and the similar and similart procedures of the type generator array (and atomic_array) will raise a failure(string) exception if the array argument is mutated in such a way as to cause a bounds exception when an element is fetched.

IV.4. Type Inclusion

Type inclusion (the new notion, see Section 6.1) is used in all contexts, including the decls of except and tagcase statements, where CLU had previously required type equality.

IV.5. Where Clauses

CLU had syntax in the where clause (specifically the production for op_name) that allowed one to require an instantiation of a type parameter's generator. This little used feature has been superseded by the mechanism described in Section 12.6.

IV.6. Uninitialized Variables

An uninitialized variable reference error is defined to cause a crash of the guardian, rather than raising a failure exception, which could conceivably be caught.

IV.7. Lexical Changes

Several new reserved words were added. In addition, the semicolon (;) was banished from the syntax.

IV.8. Input/Output Changes

The input/output data types (file_name, stream, and istream) and the library procedures described in appendix III of the CLU manual are not furnished by the Argus system. Our current implementation of Argus provides a keyboard cluster for input and a petream cluster for output. In addition, most of the built-in types currently have print operations defined, for pretty-printing objects onto petreams. These I/O mechanisms, however, are still experimental, and so are not documented in this reference manual.
Index

action 8
  built-in atomic types 9, 30, 133, 141, 146
  object 9
  type 9, 97
Atomic_array 30, 52, 133
Atomic_record 30, 52, 141
Atomic_variant 30, 64, 146
Background 8, 99
Bind 48
  and equates 50
  and routine equality 49
Block 58
Block structure 36
BNF 17, 107
Body 57
Bool 22, 54, 121
Break 63
Built-in
  atomic types 9, 30
  type 22, 119
Built-in type
  versus CLU 159
Call 4, 40, 41, 44, 50, 51, 57
  action 41
  by sharing 4, 40
  by value 4, 12, 41, 93
  creater 44, 51
  expression 50
  handler 50
  local 40
  message 43
  procedure 50
  remote 11, 41, 44, 50, 51, 89
  semantics of creator call 44
  semantics of remote call 43
  statement 57
Call action 41, 43, 44
 Candid 54
Catalog 15
Char 23, 125
  escapes 115, 23
Closure 48
CLU 3, 11, 21, 24, 73, 159
  built-in types taken from 22
  differences from 159
Cluster 77
Coarm 59
  controlling 60
Coerter 59
  foreach clause 59
Comment 20, 115
Commit 8, 10, 59, 60, 69, 88, 97
  and exception handling 73
  committed descendant 10
  of a remote call action 41
  of a subaction 9
  to the top 10
  two phase commit protocol 8, 60
Concurrency 8, 33, 39, 59
Constant 38, 47, 81
Constructor 52
  array 26, 52
  none for user-defined types 52
  record 27, 52
Abort 8, 10, 60, 61, 89, 72, 88, 97
  and exception handling 73
  of a remote call action 41
  of a subaction 9
qualifier 58, 61, 69, 72
Action 8, 56, 86, 97
  abortion versus seize statements 60
  activation action 41, 43
  ancestors 10
  and exception handling 73
  call action 41
  coauthor statement 59
  deadlock 13
  descendants 10
  divisible termination of 60
  enter statement 59
  nested 8
  nested topaction 11, 60
  orphan 12, 61
  parent of 9
  subaction 8
  termination 60, 69
  topaction 9
See also atomic
Activation action 41, 43
Actual argument 40
Actual parameter 80, 81
Ancestor 10
Any 22, 24, 32, 150
  versus CLU 159
  versus image 32
Argument
  actual 40
  versus parameter 80
Array 25, 52, 130
  constructor 26
Assignment 4, 39, 40
  and concurrency 39
  implicit 39
  multiple 39
  simple 39
  statement 39
  type checking for 39
Atomic 3, 8, 97
Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator</td>
<td>48, 62, 76, 148</td>
</tr>
<tr>
<td>bound</td>
<td>48</td>
</tr>
<tr>
<td>equality of bound iterators</td>
<td>49</td>
</tr>
<tr>
<td>type</td>
<td>148</td>
</tr>
<tr>
<td>Itertype</td>
<td>148</td>
</tr>
<tr>
<td>Keyboard</td>
<td>160</td>
</tr>
<tr>
<td>Leave</td>
<td>61</td>
</tr>
<tr>
<td>Lexicographic order</td>
<td>126, 138, 139, 141</td>
</tr>
<tr>
<td>Library</td>
<td>15</td>
</tr>
<tr>
<td>Literal</td>
<td>20, 47</td>
</tr>
<tr>
<td>char</td>
<td>115</td>
</tr>
<tr>
<td>int</td>
<td>115</td>
</tr>
<tr>
<td>real</td>
<td>115</td>
</tr>
<tr>
<td>string</td>
<td>115</td>
</tr>
<tr>
<td>Local</td>
<td>3</td>
</tr>
<tr>
<td>call</td>
<td>40, 50</td>
</tr>
<tr>
<td>object</td>
<td>7</td>
</tr>
<tr>
<td>Locking</td>
<td>9, 10, 13, 30</td>
</tr>
<tr>
<td>deadlock</td>
<td>13</td>
</tr>
<tr>
<td>for built-in atomic types</td>
<td>9</td>
</tr>
<tr>
<td>table of locking rules</td>
<td>10</td>
</tr>
<tr>
<td>Loop</td>
<td>62</td>
</tr>
<tr>
<td>Modifies</td>
<td>119</td>
</tr>
<tr>
<td>Module, 5, 75, 87</td>
<td></td>
</tr>
<tr>
<td>instantiation of</td>
<td>80, 81</td>
</tr>
<tr>
<td>parameterized</td>
<td>80</td>
</tr>
<tr>
<td>Mutable</td>
<td>3, 21</td>
</tr>
<tr>
<td>versus atomic</td>
<td>22</td>
</tr>
<tr>
<td>Mutex</td>
<td>11, 33, 98, 151</td>
</tr>
<tr>
<td>changed operation</td>
<td>99</td>
</tr>
<tr>
<td>guidelines</td>
<td>98</td>
</tr>
<tr>
<td>multiple</td>
<td>104</td>
</tr>
<tr>
<td>sharing</td>
<td>100</td>
</tr>
<tr>
<td>Name</td>
<td>35, 115</td>
</tr>
<tr>
<td>versus idr</td>
<td>35</td>
</tr>
<tr>
<td>Nested action</td>
<td>8</td>
</tr>
<tr>
<td>Nestled topaction</td>
<td>11, 80</td>
</tr>
<tr>
<td>Nil</td>
<td>22, 120</td>
</tr>
<tr>
<td>Node</td>
<td>34, 44, 120</td>
</tr>
<tr>
<td>of guardian creation</td>
<td>44</td>
</tr>
<tr>
<td>Null</td>
<td>22, 120</td>
</tr>
<tr>
<td>Object</td>
<td>3, 21, 77, 78</td>
</tr>
<tr>
<td>abstract</td>
<td>78</td>
</tr>
<tr>
<td>as value of expression</td>
<td>47</td>
</tr>
<tr>
<td>atomic</td>
<td>3, 21, 97</td>
</tr>
<tr>
<td>concrete</td>
<td>78</td>
</tr>
<tr>
<td>global</td>
<td>3, 7</td>
</tr>
<tr>
<td>immutable</td>
<td>3, 21</td>
</tr>
<tr>
<td>implementation</td>
<td>77</td>
</tr>
<tr>
<td>local</td>
<td>3, 7</td>
</tr>
<tr>
<td>mutable</td>
<td>3, 21</td>
</tr>
<tr>
<td>non-atomic</td>
<td>21</td>
</tr>
<tr>
<td>references</td>
<td>3</td>
</tr>
<tr>
<td>representation</td>
<td>77</td>
</tr>
<tr>
<td>sharing</td>
<td>3, 98, 100</td>
</tr>
<tr>
<td>stable</td>
<td>3, 7</td>
</tr>
<tr>
<td>transmissible</td>
<td>3, 12, 21, 93</td>
</tr>
<tr>
<td>transmission of cyclic</td>
<td>96</td>
</tr>
<tr>
<td>objects</td>
<td>3</td>
</tr>
<tr>
<td>versus variable</td>
<td>3</td>
</tr>
<tr>
<td>volatile</td>
<td>7</td>
</tr>
<tr>
<td>Oneof</td>
<td>63, 143</td>
</tr>
<tr>
<td>Obplooding</td>
<td>81</td>
</tr>
<tr>
<td>Operation</td>
<td>77</td>
</tr>
<tr>
<td>indivisibility</td>
<td>21, 119</td>
</tr>
<tr>
<td>Operator</td>
<td>20</td>
</tr>
<tr>
<td>binary</td>
<td>53</td>
</tr>
<tr>
<td>infix</td>
<td>53</td>
</tr>
<tr>
<td>precedence</td>
<td>54</td>
</tr>
<tr>
<td>prefix</td>
<td>53</td>
</tr>
<tr>
<td>unary</td>
<td>53</td>
</tr>
<tr>
<td>Optional parameter</td>
<td>82, 84</td>
</tr>
<tr>
<td>Orphan</td>
<td>12, 44, 61</td>
</tr>
<tr>
<td>Overview</td>
<td>119</td>
</tr>
<tr>
<td>Own data</td>
<td>46, 85</td>
</tr>
<tr>
<td>Own variable</td>
<td>85</td>
</tr>
<tr>
<td>and crash recovery</td>
<td>85</td>
</tr>
<tr>
<td>Parameter</td>
<td>47, 80</td>
</tr>
<tr>
<td>actual</td>
<td>81</td>
</tr>
<tr>
<td>optional</td>
<td>82</td>
</tr>
<tr>
<td>versus argument</td>
<td>80</td>
</tr>
<tr>
<td>Parameterization</td>
<td>80</td>
</tr>
<tr>
<td>Parameterized type</td>
<td>21, 81</td>
</tr>
<tr>
<td>instantiation</td>
<td>81</td>
</tr>
<tr>
<td>Parent</td>
<td>9</td>
</tr>
<tr>
<td>Pause</td>
<td>86</td>
</tr>
<tr>
<td>Post</td>
<td>119</td>
</tr>
<tr>
<td>Pragmas</td>
<td>153</td>
</tr>
<tr>
<td>Pre</td>
<td>119</td>
</tr>
<tr>
<td>Precedence</td>
<td>54</td>
</tr>
<tr>
<td>Principal argument</td>
<td>30</td>
</tr>
<tr>
<td>Print</td>
<td>160</td>
</tr>
<tr>
<td>Private routine</td>
<td>78</td>
</tr>
<tr>
<td>Procedure</td>
<td>48, 75, 148</td>
</tr>
<tr>
<td>bound</td>
<td>48</td>
</tr>
<tr>
<td>closure</td>
<td>48</td>
</tr>
<tr>
<td>equality of bound procedures</td>
<td>49</td>
</tr>
<tr>
<td>type</td>
<td>148</td>
</tr>
<tr>
<td>Process</td>
<td>8, 59</td>
</tr>
<tr>
<td>See also action</td>
<td></td>
</tr>
<tr>
<td>Proctype</td>
<td>148</td>
</tr>
<tr>
<td>Punctuation</td>
<td>180</td>
</tr>
<tr>
<td>Punctuation token</td>
<td>20</td>
</tr>
<tr>
<td>Qualifier</td>
<td></td>
</tr>
<tr>
<td>assert</td>
<td>56, 61, 89</td>
</tr>
<tr>
<td>action, topaction</td>
<td>59</td>
</tr>
<tr>
<td>Raise</td>
<td>70</td>
</tr>
<tr>
<td>Read lock</td>
<td>9</td>
</tr>
<tr>
<td>Reader</td>
<td>30</td>
</tr>
<tr>
<td>Real</td>
<td>23, 123</td>
</tr>
<tr>
<td>Record, 52, 139</td>
<td></td>
</tr>
<tr>
<td>constructor</td>
<td>27</td>
</tr>
<tr>
<td>Recover code</td>
<td>8, 89</td>
</tr>
<tr>
<td>Recoverable</td>
<td>8, 97, 98</td>
</tr>
<tr>
<td>Recovery</td>
<td>8, 89, 87</td>
</tr>
<tr>
<td>Refer</td>
<td>3</td>
</tr>
<tr>
<td>Reference</td>
<td>34, 47</td>
</tr>
<tr>
<td>Remote call</td>
<td>11, 41, 44, 50, 51, 89</td>
</tr>
<tr>
<td>semantics</td>
<td>43</td>
</tr>
<tr>
<td>Replicated database example</td>
<td>60</td>
</tr>
<tr>
<td>Representation</td>
<td>77</td>
</tr>
<tr>
<td>concrete</td>
<td>78</td>
</tr>
<tr>
<td>external</td>
<td>12, 94</td>
</tr>
<tr>
<td>Required operation</td>
<td>81</td>
</tr>
<tr>
<td>Reserved operation</td>
<td>81</td>
</tr>
<tr>
<td>Required operation</td>
<td>81</td>
</tr>
<tr>
<td>Resignal</td>
<td>72</td>
</tr>
<tr>
<td>Resilience</td>
<td>97, 98</td>
</tr>
<tr>
<td>See also recoverable</td>
<td></td>
</tr>
<tr>
<td>Restriction</td>
<td>80, 81</td>
</tr>
</tbody>
</table>
Result 47
Return 61
Routine 75, 76, 90
equality 83
See also iterator, procedure
RPC
See also remote call
Rules 193
Scope 36, 78
rules 35
unit 35
Seize 66, 98
Selection
of component 51
of element 51
Self 48, 88
Separator 19, 20, 115
Sequence 25, 52, 128
constructor 25
Serializable 8, 9, 67, 97
Set_operation 56
Sharing 3
and mutex 103
and transmission 96
Signal 69
See also exception
Spooler guardian 90
Stable
object 3, 7
state 8, 87
storage 8, 97
storage and closures 49
storage recovery 89
variable 3, 87
See also resilience
Statement 57
abort break 63
abort continue 63
abort leave 61
abort prefix 59
abort resultant 72
abort return 61
abort signal 69
assignment 39
block 58
break 63
counter 59
component update 58
conditional 62
continue 63
count 57
element update 58
enter 59
except 70
exit 72
for 62
fork 58
if 62
iteration 62
leave 61
pause 66
resultant 72
return 61
seize 66
signal 69
tagcase 63
taglast 64
tagwait 65
termiate 67
update 58
while 62
yield 62
Store operation 58
String 24, 126
See also character escapes
Struct 26, 62, 136
constructor 27
Structure
See also struct
Subaction 8, 10, 41, 59
Synchronization 38, 97
Synchronous 90
Syntax 107
Table example, transmission of 95
Tagcase 63
Taglast 64
Tagwait 65
Terminate 67
Termination
exceptional 69
of a guardian 67, 90
of a routine 40
Then 82
Token 19, 115
Topaction 9, 59
nested 11
Transmittable 3, 12, 21, 93
object 12
Transmit 21, 41, 78, 84, 93
actual 84
for parameterized modules 94
True 22, 121
Two-phase commit 8, 50, 60, 73
Type 3, 4, 15, 21, 39, 77, 81
actual 81
atomic 9, 97
built-in 22, 119
built-in atomic types 9
correctness 4
equality 83
external representation 12, 94
generator 21, 80, 81
guardian interface 31
implementation of 77
inclusion 4, 22
of a creator 32, 149
of a guardian 31
of a handler 32, 149
of a iterator 148
of a procedure 148
parameter 34, 81
parameterized 9, 21, 80
safety 4
set 80
transmittable 12, 21, 93
user-defined 34, 52, 77
versus type actual 82
See also cluster, guardian
Type checking 15, 39, 83
of an instantiation 83
Type inclusion 4, 22
versus CLU 160
Type_spec 21
Index

Unavailable 11, 42, 43, 44, 59, 60
Unhandled exception 73
  versus CLU 159
Uninitialized variable 36
  versus CLU 160
Up 55, 78
Update statement 58
Value 47
Variable 3, 36, 47
  own variable 85
  stable 3, 97
  uninitialized 36
  versus object 3
Variant 63, 144
Version
  of an atomic object 9
Volatile
  object 7
  state 8, 87
  variable 87
Where clause 80, 160
  subordinate 82
While 62
Write lock 9
Writer 30
Yield 62
This blank page was inserted to preserve pagination.
Report #: LCS-TR-400

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intelligence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR)  ☐ Technical Memo (TM)
☐ Other: __________________________

Document Information

Number of pages: 175 (181 images)

Not to include DOD forms, printer instructions, etc... original pages only.

Intended to be printed as:

☐ Single-sided or
☒ Double-sided

Print type:

☐ Typewriter  ☐ Offset Press  ☐ Laser Print
☐ InkJet Printer  ☐ Unknown  ☐ Other: __________________________

Check each if included with document:

☒ DOD Form (2)  ☐ Funding Agent Form  ☐ Cover Page
☐ Spine  ☐ Printers Notes  ☐ Photo negatives
☐ Other: __________________________

Page Data:

Blank Pages (by page number): 14, 16, 18, 46, 56, 68, AND PAGES FOLLOWING TITLE PAGE ARE BLANK

Photographs/Tonal Material (by page number): __________________________

Other (note description/page number):

Description: Page Number:

IMAGE MAP: (1-175) UNITLE PAGE, UNHVD BLANK,

1-VII, UNHVD BLANK, 1-165

(176-181) SCANCONTROL, DOC (2), TEST (3)

Scanning Agent Signoff:

Date Received: 4/26/96  Date Scanned: 4/29/96  Date Returned: 5/2/96

Scanning Agent Signature: __________________________
**REPORT DOCUMENTATION PAGE**

1a. REPORT SECURITY CLASSIFICATION: Unclassified

2a. SECURITY CLASSIFICATION AUTHORITY: MIT/LCS/TR-400

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE: Approved for public release; distribution is unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
MIT Laboratory for Computer Science

6b. OFFICE SYMBOL (If applicable)

7a. NAME OF MONITORING ORGANIZATION
Office of Naval Research/Department of Navy

7b. ADDRESS (City, State, and ZIP Code)
Information Systems Program
Arlington, VA 22217

6c. ADDRESS (City, State, and ZIP Code)
545 Technology Square
Cambridge, MA 02139

8a. NAME OF FUNDING/SPONSORING ORGANIZATION
DARPA/DOD

8b. OFFICE SYMBOL (If applicable)

10. SOURCE OF FUNDING NUMBERS

11. TITLE (Include Security Classification)
Argus Reference Manual

12. PERSONAL AUTHOR(S)
Liskov, Barbara; Day, Mark; Herlihy, Maurice; Johnson, Paul; Leavens, Gary (editor); Scheifler, Robert; and Weihi, William

13a. TYPE OF REPORT
Techniocal

13b. TIME COVERED
FROM TO

14. DATE OF REPORT (Year, Month, Day)
1987 November

15. PAGE COUNT
165

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Distributed systems, fault-tolerance, nested transactions, concurrency, concurrency control, locking, persistent storage, abstract data types, atomic objects, remote (cont.)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Argus is an experimental language/system designed to support the construction and execution of distributed programs. Argus is intended to support only a subset of the applications that could benefit from being implemented by a distributed program. Two properties distinguish these applications: they make use of on-line data that must remain consistent in spite of concurrency and hardware failures, and they provide services under real-time constraints that are not severe. Examples of such applications are office automation systems and banking systems.

Argus is based on CLU. It is largely an extension of CLU, but there are number of differences. Like CLU, Argus provides procedures for procedural abstraction, iterators for control abstraction, and clusters for data abstraction. In addition, Argus provides guards that encapsulate and control access to one of more resources. Argus also provides equate and abstraction as a convenient way to refer to constants. As in CLU, modules may be parameterized, so that a single module can define a class of related abstractions.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL
Judy Little, Publications Coordinator

22b. TELEPHONE (Include Area Code)
(617) 253-5894

22c. OFFICE SYMBOL

SECURITY CLASSIFICATION OF THIS PAGE

Unclassified
18. procedure call, orphans, exception handling.