
Robert E. Strom
David F. Bacon    Arthur P. Goldberg    Andy Lowry
Daniel Yellin

IBM T. J. Watson Research Center

June 27, 1990

1PO Box 704, 30 Saw Mill River Road, Yorktown Heights, NY 10598
Hermes *Greek Mythology*. The god of commerce, invention, cunning, and theft, who also served as messenger and herald for the other gods, as patron of travelers and rogues, and as the conductor of the dead to Hades; identified with the Roman god Mercury.\(^1\)

\(^1\)([Mor80])
Preface

This document contains a tutorial, a reference manual, and some appendices.

The tutorial introduces you to Hermes by guiding you through a set of examples. We begin with a simple program which outputs “Hello world”, and continue through more complex examples, ending with a window system. We then discuss additional useful features of Hermes, including an important innovation in static checking—typestate analysis. We assume that you have some experience writing application programs in a high-level procedural language, such as C, Pascal, or Ada. We will highlight the differences between programming in Hermes and programming in the other languages. This way you will get a feeling for “idiomatic Hermes”. At the end of the tutorial, you will know the basic vocabulary of Hermes. You will be able to write some Hermes programs by imitating the examples. You will be able to compare Hermes to other languages. But you will not know the precise rules of Hermes—these are covered in the reference manual.

The reference manual is more formal than the tutorial. We also give examples in the reference manual, but with a different purpose. The examples in the reference manual illustrate the language rules. They highlight the difference between legal and illegal programs rather than illustrate “typical” programs.

The appendices are the most formal. They contain the rules of Hermes in tabular form. They are produced from the same machine-readable files that are used to produce the compiler itself.

This document does not describe how to use any of the existing Hermes implementations, nor does it describe their idiosyncrasies. Such information is to be found in the *Hermes Users Guide* distributed with the implementation.

Readers wishing to obtain an experimental working Hermes system should send electronic mail to hermes-request@ibm.com or U.S. mail to one of the authors at the address listed on the title page. Hermes currently (March 1990) runs on Sun 3 and Sun 4 systems running SunOS, and IBM RTs running Berkeley 4.3 Unix.
# Contents

## Preface

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>ii</td>
</tr>
</tbody>
</table>

## I Tutorial

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Introduction to Hermes</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Getting Started—A Simple Hermes Program</td>
<td>5</td>
</tr>
<tr>
<td>1.3 A Second Program</td>
<td>10</td>
</tr>
<tr>
<td>1.4 Putting Processes Together</td>
<td>12</td>
</tr>
<tr>
<td>1.5 Declarations and Definitions</td>
<td>18</td>
</tr>
<tr>
<td>1.5.1 Declarations</td>
<td>18</td>
</tr>
<tr>
<td>1.5.2 Definitions</td>
<td>21</td>
</tr>
<tr>
<td>1.6 A Simple Server</td>
<td>25</td>
</tr>
</tbody>
</table>

## 2 A Miniature System

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Requirements</td>
<td>31</td>
</tr>
<tr>
<td>2.2 Design</td>
<td>32</td>
</tr>
<tr>
<td>2.3 Interfaces</td>
<td>33</td>
</tr>
<tr>
<td>2.4 Window System Shell</td>
<td>36</td>
</tr>
<tr>
<td>2.5 Front-end Process</td>
<td>37</td>
</tr>
<tr>
<td>2.6 Tokenizer Procedure</td>
<td>41</td>
</tr>
<tr>
<td>2.7 The Window Manager</td>
<td>43</td>
</tr>
<tr>
<td>2.7.1 Definitions</td>
<td>43</td>
</tr>
<tr>
<td>2.7.2 Skeleton</td>
<td>44</td>
</tr>
<tr>
<td>2.7.3 Refocussing and Writing Output</td>
<td>45</td>
</tr>
<tr>
<td>2.7.4 Dispatching Lines to a Particular Window</td>
<td>45</td>
</tr>
<tr>
<td>2.7.5 Creating and Killing Windows</td>
<td>47</td>
</tr>
<tr>
<td>2.8 Creating a Window Application</td>
<td>48</td>
</tr>
<tr>
<td>2.8.1 Application Builder</td>
<td>49</td>
</tr>
<tr>
<td>2.8.2 Adapter and Quit Dispatcher</td>
<td>51</td>
</tr>
<tr>
<td>2.9 Summary</td>
<td>53</td>
</tr>
</tbody>
</table>
3 Type and Typestate Checking

4 Additional Hermes Constructs
4.1 Expression Blocks .................................. 59
4.2 Send ............................................. 60
4.3 Variants .......................................... 61
4.4 Polymorphs ....................................... 64

5 Research Directions in Hermes

II Hermes Reference Manual

6 Introduction

7 Lexical and Syntactic Rules

8 Resolution
8.1 Variable Names .................................... 75
8.1.1 Base Variables ................................. 76
8.1.2 Component Names .............................. 77
8.2 Type Names ....................................... 77
8.3 Attribute Names .................................. 78
8.4 Exception Names ................................ 78
8.5 Exit Names ....................................... 79
8.6 Process Names ................................... 79

9 Type Checking and Inference

10 Typestate Checking
10.1 Syntax of Typestates ............................. 84
10.2 Formal Typestates ............................... 85
10.3 Valid Typestates ................................ 86
10.4 Ordering of Typestates ......................... 87
10.5 Coercions ....................................... 87
10.6 Constants ....................................... 88
10.7 How to Apply the Typestate Rules ............ 88
10.8 The Checking Algorithm ......................... 90
10.9 Typestate Errors ................................. 91

11 Hermes Operations
11.1 Ubiquitous Operations .......................... 93
11.2 The Depletion Exception ....................... 95
11.3 Control Flow Operations ....................... 96
11.4 Scalar Operations ............................... 101
11.5 Record Operations ............................. 107
11.6 Table Operations ........................................ 107
11.7 Variant Operations ..................................... 115
11.8 Process Creation and Communication Operations ....... 118
11.9 Polymorph Operations .................................. 125
11.10 Program Operations .................................... 128
11.11 Constraints ............................................. 131

A Hermes Concrete Syntax ..................................... 134
  A.1 Lexical Rules ............................................. 135
  A.2 Syntactic Rules .......................................... 138

B Hermes Operations ........................................... 150
  B.1 Operation Descriptions ................................ 150
    B.1.1 Description Header .................................. 151
    B.1.2 Type Rules .......................................... 151
    B.1.3 Preconditions ....................................... 152
    B.1.4 Postconditions ..................................... 152
    B.1.5 Special Rules ...................................... 153
    B.1.6 Operation Semantics ................................ 153
  B.2 Type Classes ............................................ 153
  B.3 Inference Functions ..................................... 155
  B.4 Precondition Functions .................................. 155
    B.4.1 Typestate Preconditions .......................... 155
    B.4.2 Context Preconditions ............................. 157
    B.4.3 Conditional Exceptions ............................. 158
  B.5 Postcondition Functions ................................ 158
  B.6 Operation Descriptions ................................ 160

C Predefined Module ........................................... 194
  C.1 References ............................................... 211
Part I

Tutorial
1

Introduction to Hermes

1.1 Introduction

Hermes is an experimental language developed at the IBM TJ Watson Research Center. It reflects our research group's opinion of how complex systems should be programmed ([BS89, SY83, PS83, SYB87c]).

The basic idea behind Hermes is to take as much work as possible away from the programmer and give it to the compiler implementor. Hermes is a simple language for expressing computation and program composition. The Hermes compiler chooses the implementation instead of you. For example, if you wish to build a list of names and addresses, you just declare a table of name-address pairs. You don't worry about whether this table is stored in the computer as an array, a linked list, a hash table, a splay tree, or a disk file—that is the compiler's job. Instead of the approximately 120 system calls in a typical UNIX ([KP84]), you have just 10 Hermes statements which deal with creating programs, connecting them together, and communicating between them. You don't worry about distribution, communication, or coping with failures. With just these ten statements, you can build arbitrary systems, including some which in UNIX would require superuser privilege to build.

Of course, a lean language isn't all that's required. After all, the Turing machine language is even leaner! Hermes also stresses the ability to build systems out of separately developed, compiled, and tested modules. We expect non-interference between modules — this means that a module should behave the same when combined with other modules into a larger system as it did when tested in isolation. We want as much compile-time checking as possible to detect nonsensical programs and to detect interface mismatches. In exchange for modularity and checkability we are willing to require programmers to write more documentation, e.g., declarations and interface definitions.

How does this affect you, the programmer? We are assuming that you are used to programming in a language like C, Pascal, or Ada, under an operating system like UNIX, MS-DOS, or VMS. Will you have to completely change your concept of programming?

Not really. In many respects, Hermes is a traditional imperative language. Hermes has variables, assignment statements, and the usual sequential control structures. Although Hermes is explicitly designed to run in concurrent and distributed environments, you do not need to write parallel execution statements (e.g., cobegin), semaphores, monitors, guardians,
or transactions. A Hermes module is a straightforward sequential program with inputs, outputs, and local memory. A Hermes system is built by taking several Hermes modules and connecting the output of some modules to the input of others.

But some aspects of Hermes are very different from what a traditional systems programmer is used to:

- There are no pointers. No address variables, no address arithmetic, no addresses period. You have no control of how data is stored in memory. That means that bit order, byte order, alignment, and all other such details which may vary from machine to machine are invisible to you.

- There is no shared data. Not via pointers, since there are no pointers, not via nesting of inner procedures within outer procedures, not via global or external declarations of any kind. Every variable belongs to exactly one module. There is no aliasing—that is, two distinct variable names always denote two distinct variables.

- You will use many processes. Hermes is a process-oriented language. This means that Hermes processes take on the role that modules have in procedural languages. You wouldn’t dream of building a system with thousands of UNIX processes, but you may well build a system with thousands of Hermes processes. Most of the time, communication between Hermes processes is at least as fast as a procedure call. But even though there are many processes, there is no shared data—only queued communication. So you can still analyze each process independently and view it as a sequential program.

- There is no language vs. system duality. When you program in C, you have to think about the C world and the UNIX, MS-DOS or VMS world. In the C world, data is stored in typed variables such as scalars, arrays, structs, or pointers, and passed between procedures using CALLs. In the operating system world, data is stored in buffers or in files, and passed from one application to another using pipes, sockets, or files. Data is untyped—it is interpreted as strings of bytes. Some data (e.g. pointers, file handles) cannot be passed to other processes or stored in files. This data must be marshalled and demarshalled. In distributed programs, you must be aware of what is local, what is remote, and often what kind of machine you’re running on. In Hermes, there is only one world to think about—the Hermes world.

- Hermes has much more compile-time checking ([SY86]). Much more than in C, and somewhat more even than Ada, Modula-3, or other newer languages. The benefits: (1) more errors are discovered earlier, and (2) different users can safely run in the same address space, even
though some programs may have bugs and the users don’t trust one another. There is some cost: you have to write more declarations.

1.2 Getting Started—A Simple Hermes Program

Let’s start with our first Hermes program.

```hernes
receive Parms from Init;
call Parms.PutLine("Hello, World!");
return Parms;
```

Before you compile this program, you have to add declarations and possibly type definitions. We’ll show you in a later section how to write them. Let’s assume for now that you have already written them or that they have already been supplied for you.

This program can be used to display the message Hello, World!. How does it differ from the corresponding program in C, Ada ([SYW85]), or Pascal?

```c
printf("Hello, World!"); /* a C program to do the above */
```

```ada
PUT("Hello, World!") -- An ADA program
```

```pascal
write("Hello, World!") {A Pascal program }
```

Although at first glance the programs appear to differ only in superficial syntax, there is nevertheless an important distinction between the Hermes version of this simple program and all the others—the late binding of `PutLine`. In Hermes, `PutLine` is a parameter; its counterparts in the other languages are constants.

In the C program, `printf` has a fixed binding—it denotes a library program which writes the character string to the standard output. The only way to change the binding is to link your C program to a different library. If you do this, you will also change the binding of `printf` for all the modules of your program. To use the terminology of the C reference manual ([KR78]), functions are not variables. If you want to control the binding of a function with a C program, you must simulate function variables using pointers to functions. (Similarly in Pascal ([Coo83]) and Ada, functions are not variables. In these languages there is a way to achieve a limited degree of dynamic binding.)

So even in this tiny Hermes program, you can see two important features of Hermes: (1) nothing is global, and (2) every function or procedure name refers to a variable whose value is determined at run-time, not compile-time.

Why do we do this? After all, the program is longer because we must receive the parameter list `Parms` and because we must refer to the function as `Parms.PutLine`. Hermes is designed this way because it assumes you
are writing a multi-user or multi-application system rather than a single application.

Look at figure 1.1, which depicts a traditional system such as UNIX. A kernel maintains multiple address spaces in which you run your application programs. Your C application is a program running in a single address space. Resources outside the address space (e.g. terminals and printers) are managed by the operating system, and assigned to the address space as a whole. It makes sense for all the `printf` statements within an application to have a single global meaning, because “global” means within the address space. Joe's application and Jane's application can each have different bindings to their respective `printf` functions.

But suppose my application is an entire system, which may include Joe's application, Jane's application, modules available to both users, and modules written by one user but accessible to some modules of another. (See, for example, figure 1.2.) In this situation, it is clear that no function name can be global to the whole system. Although Joe's application and Jane's application may both be part of a single large “program”, it does not make
FIGURE 1.2. A multi-application system viewed as a single high-level language "program"
sense for Joe's `printf` statement to denote the same console as Jane's.

Hermes encourages you to think of systems from the viewpoint of figure 1.2, rather than figure 1.1. The differences are: (1) the distinction between intra-user and inter-user communication disappears; (2) access to resources is controlled at the level of individual modules, rather than address spaces; and (3) access control decisions are made by programs, not by a fixed kernel.

Given that Hermes avoids static binding and global variables, how is the binding done? Program `HelloWorld` illustrates the simplest way of doing the binding—making the function `PutLine` a parameter to the program.

This is a good time to begin using the Hermes terminology. The source code for program `HelloWorld` is called a process module. A process module can be instantiated to create a process—an active entity with state, which executes the statements of the process module. `Parms`, `Parms .PutLine`, and `Init` are variable names.

Every variable name has a statically known type. A variable's type determines what operations on the variable are legal and what types the other operands must have. Types are organized into type families based on the operations they allow. For example, `Parms .PutLine` is of type family output port. Output ports support the operation call. The value of an output port is a connection to an input port, which is a message queue in another process.

Program `HelloWorld` communicates with its output process by executing the statement `call Parms .PutLine`. The call parameters (in this case, the single parameter "Hello World") are bundled into a callmessage, which is then queued on the input port to which the output port is connected. The calling process waits until the callmessage is returned.

The type of an output port determines the type of input port it can connect to. The type of an input port, in turn, determines the type of callmessage which can be sent to it. In this example, the input port expects a callmessage with a single character string parameter. The call statement's argument list must contain a single character string. A call with the wrong number or types of arguments will be rejected at compile time as a type error.

When program `HelloWorld` starts up after being instantiated, the only initialized variable is the input port named `Init`, which was designated as the initialization port. By convention, the creator of a process— which we call the parent — sends the newly created process — the child — a callmessage over its initialization port. The callmessage contains the parameters the child needs to get started, including whatever ports it may need to communicate with the outside. In this example, we assume that `HelloWorld`'s parent—perhaps a shell process—passes a callmessage with parameters named `PutLine`, `GetLine`, `GetProgram`, and `ParmString`. `GetLine` and `GetProgram` are ports to services to obtain input lines and load programs, and `ParmString` is a command string. These other parameters aren't used
by the HelloWorld program, but they will be used in other examples later, and we are assuming a common interface between the shell and its children. It's easy to write a shell in Hermes. In a later section, we will see how to write one.

The receive statement dequeues the callmessage from the input port Init and stores it in the callmessage variable Parms. Now the parameters, named Parms.PutLine, Parms.GetLine, etc., are initialized. The statement call Parms.PutLine is legal here. It would have been illegal to call Parms.PutLine before the receive statement, because Parms.PutLine is uninitialized at the beginning of the program. Such a call would be rejected at compile time as a typestate error. Typestate checking ([SY86]) is a form of static checking which is new to Hermes and will be discussed in detail later.

The call statement creates a callmessage with a single string component with value "Hello, World!". This message is sent to the input port at the other end of the connection Parms.PutLine. Which port is that? It depends upon the value of parameter PutLine which was passed by the parent process. We don't know which port it is, and we don't care. It may be a service which writes the line on an output device. Or the port may belong to a Hermes process which filters the output and passes it along to another port. When we look at the program in isolation, we care only that the caller and receiver agree upon what is sent and returned in the call. In this case, the interface will say that the parameter type must be a character string and the parameter typestate must be initialized. Hermes guarantees that Parms.PutLine will be bound only to input ports expecting to receive callmessages containing a single initialized character string parameter.

After the call completes, program HelloWorld returns the callmessage in Parms using a return statement. The callmessage is returned to the process which originally made the call—in this case, the parent process. After the return of the callmessage, variable Parms becomes uninitialized. It would be illegal to call Parms.PutLine here. Since there are no more statements, the process terminates.

Because Hermes is a process-oriented language, it will be less confusing if you think of making a call and returning a call as sending and returning a callmessage, rather than as transfer of control. Think of both the caller and receiver as active processes. The caller sends a callmessage and waits for the answer. The receiver receives the callmessage, processes it, possibly stores some return values into the callmessage, and then returns it.¹

Program HelloWorld is typical of what we call a client program. A

¹Note that a Hermes program terminates when it reaches the end, not when it executes the return statement. In other languages, where a return statement can be viewed as a transfer of control, execution of a return statement terminates the procedure. But a Hermes process may possibly continue execution after issuing return. We will see some examples of this later.
client corresponds to an applications program on conventional operating systems. Client programs have a specific job to do—they’re not interested in creating processes, or binding output ports. They are content to let their parent process—typically a shell or system builder—do the binding for them. Also, they are users of services—not providers of services. Roughly speaking, clients make calls, servers receive calls, and shells create and connect other processes. Some programs may play multiple roles, e.g. both a server and a client. Clients are the most common type of process and the easiest to write. In later sections, we’ll see how to use Hermes to write shells and servers.

Summary: In this section, you have seen that in Hermes, there is no global data, and in particular function names (which Hermes calls output ports) are variables, not constants. You have seen that this is a consequence of the fact that a Hermes program is an entire dynamic system, not simply your application. You have learned the following Hermes nomenclature: process module, process, variable, type, type family, typestate, input port, output port, callmessage, initialization port, initialized, uninitialized, client, server, and shell. You have seen the structure of a typical client program: first receive a parameter callmessage from the initialization port; then use the output ports passed in that callmessage to access services. You have learned the following Hermes statements: receive, call, and return.

1.3 A Second Program

Before moving on to the systems programming constructs in Hermes, let’s look at one more client program, which we shall call Echo:

```plaintext
receive Parms from Init;
block
  declare
    Line: Charstring; -- line read from standard input
  begin
    while ('true') repeat
      call Parms.GetLine(Line);
      call Parms.PutLine(Line);
    end while;
    on (GetLineInterface.EndStream)
    end block;
return Parms;
```

We assume that the initialization port Init is of the same type as in program HelloWorld. Therefore, the variable Parms will have the same component names and types as in program HelloWorld. An input port type, together with its associated callmessage type is called an interface,
so we can say that program Echo and program HelloWorld have the same initialization interface. This interface is shown pictorially in figure 1.3. It is written out in full in section 1.5.

GetLine and PutLine both take a single string-valued argument. However, GetLine assumes that its argument is *uninitialized* prior to the call, and that an initialized result is passed back on return. PutLine, on the other hand, assumes that its argument is *initialized* prior to the call. All these assumptions are recorded in the interface definition. Both clients and servers are checked at compile-time to make sure that the code agrees with the interface with respect to both type and typestate. Type checking of interfaces may be familiar from other languages, such as Ada and Pascal. Typestate checking is new. Every call interface has a pre-call and post-call typestate. The compiler checks that a caller puts each argument in the correct pre-call typestate. The compiler checks that a receiver puts each parameter in correct post-call typestate prior to return. In this example, GetLine's interface definition specifies that the call argument is uninitialized before call and initialized after return. If the compiler can't deduce that a program receiving a call will always initialize its parameter, it will generate an error message and refuse to compile the program.

GetLine's interface also defines an *exception* named EndStream. When the server is called to get a line and there are no more lines because the end of the stream has been reached, the server will return the callmessage with the EndStream exception. For each exception, there may be a different post-call typestate specified in the interface. In this example, the interface will specify that the argument to GetLine will not be initialized in the event of an EndStream exception.

The structure of program Echo is an infinite loop, calling GetLine to obtain a string and store it into variable Line, and then calling PutLine to put out the string. When the EndStream exception is raised, control jumps to the nearest matching exception handler clause in an enclosing block statement. A clause is simply a set of statements—in this case, the set is empty, so the block simply terminates, and the return follows.

Exception handling is very important in a language like Hermes. Primitive statements may raise exceptions; user-defined operations implemented via calls may raise user-defined exceptions. In some languages, a run-time error such as an overflow or division by zero causes the entire program to halt and an error message to be printed. In Hermes this is obviously unacceptable—remember the program isn't just your application, but the whole system. Return codes such as are used in C have the disadvantage that you might fail to test them. A Hermes exception can't be ignored because control jumps to a new location.

Typestate checking guarantees that you can't use a variable like Line after an exception return. The compiler will know that Line is uninitialized because GetLine's interface definition specifies that its argument will be returned uninitialized when the EndStream exception is returned. This
degree of checking goes beyond what even the most modern commercially available languages (as of 1989) offer.

Incidentally, the call to `GetLine` may also be written:

```pascal
Line := Parms.GetLine();
```

Whenever a call has `n` arguments, and the last argument is a return value, the call may be rewritten using the familiar function notation. Like other expressions, functions return anonymous values which may themselves be used as operands. In this case, the result is simply assigned to the variable `Line`, so there is no particular advantage to the function notation.

**Summary:** You have learned that calls are processed by exchanging a callmessage. (This is called passing parameters by *value-result.*) Interface definitions define which parameters passed as call arguments from the caller and which parameters returned to the caller are expected to be initialized. These expectations are enforced by typestate checking. You have learned about exceptions. You have learned the following Hermes terminology: interface, typestate checking, exception, exception handler clause. You have seen examples of the following Hermes statements: assignment, function call, `while, block`.

## 1.4 Putting Processes Together

So far, we have written simple one-process programs. These programs assumed that all ports were bound by the parent and passed as parameters in the initialization callmessage. Now you're going to learn to do your own systems programming. We will write a process that will instantiate two other processes and bind the output port of one to the input port of the second.

The interface we have been using for programs `HelloWorld` and `Echo` is depicted in figure 1.3. These programs are initially passed three ports: `PutLine`, `GetLine`, and `GetProgram`, so they are depicted as boxes with three arrows representing three connections to the environment.

We're going to build a composite program to the same interface. It will consist of our client program (either program `HelloWorld` or program `Echo`—it doesn’t matter), connected to a *filter* program. The filter has one
input port and one output port, as shown in figure 1.4. The filter program receives a call as if it were a PutLine service. Each time it is called on its input port with a line as parameter, it calls its output port passing the original line preceded by a character string called Prefix. The filter is initially passed its output port and the prefix string.

The composite program is a program with the same interface as in figure 1.3. It creates the original client, the filter, and connects them together as shown below in figure 1.5. Since the composite program has the same interface as the original single client process, it can be further composed.

We make an analogy between Hermes processes and electrical components with plugs and sockets. We can think of the process in figure 1.3 as a component with 3 plugs, and the process in figure 1.4 as a box with one plug and one socket. The combination in figure 1.5 is a box with the same three plugs, and can therefore be plugged into the same configurations as the first box. The interface corresponds to the shape of the plugs; type checking guarantees that a round plug is never plugged into a square socket. Unlike the pipes, files and sockets of traditional operating systems, the data is not restricted to being byte streams in contiguous buffers—it can be any datatype supported by the language.

Let’s look at the filter program first:

```haskell
-- initialization
receive Parms from Init;
Out := Parms.PutLine;
Prefix := Parms.Prefix;
```
new In;
connect Parms.ClientPutLine to In;
return Parms;
-- main loop
while ('true') repeat
    receive PutLineCM from In;
    call Out(Prefix | "": " | PutLineCM.Line);
    return PutLineCM;
end while;

The filter process has two sections. The first section is executed just once after the parent instantiates the filter and calls the initialization port. The second section is executed repeatedly every time a callmessage arrives on the filter’s In input port—this will happen when the client makes a call on its PutLine output port.

Let’s begin with the initialization. The filter’s interface is different from the one we have seen in HelloWorld and Echo. Those processes’ parents passed three output ports and a string. The filter’s parent will pass one output port—the connection to the PutLine service—as Parms.PutLine, and a prefix string as Parms.Prefix. It expects to receive back an output port, Parms.ClientPutLine, connected to the filter’s input port In.

The initialization code first copies Parms.PutLine and Parms.Prefix into local variables. This is necessary because the callmessage Parms will no longer be initialized after the return statement. Unlike the previous examples, the filter has real work to do after returning the initialization call—it must service PutLine calls from its client.

After saving the two variables it needs, it then initializes its input port In. This is done with the new statement. It then assigns the output port variable Parms.ClientPutLine a connection to the input port In, using a connect statement. The connect statement is the Hermes primitive for creating connections between output ports and input ports. It now returns the initialization callmessage. This callmessage will contain an output port connected to the filter’s input port In.

Recall that the return statement returns a message; it does not “transfer control” as it would in a language without multiple processes. The return statement marks the end of the initialization section of the filter, and the beginning of its main section.

The main job of the filter is to receive calls on input port In, and reissue these calls with transformed data. We code this as an infinite while loop. At each iteration of the loop, we dequeue a single callmessage into variable PutLineCM. Then we service the call by calling the ‘real’ PutLine service with an argument computed by prefixing the original string with a string prefix and a colon. Finally, we return the original callmessage. The filter will run indefinitely until its connection is severed. We’ll see in a later section how this is done.

Now let’s look at the code of the parent process, Compound. This process
creates the client, creates the filter, and connects the two together. The client may be any program with our standard 3-plug interface, such as HelloWorld, Echo, or even Compound itself. We'll assume that the input string ParmString received by Compound contains the name of the client program, and its arguments, separated by the delimiter '/'.

```python
receiveParms from Init;
block begin
    ClientName := "filter";
    Filter := create of ParmString.GetProgram(ClientName);
    Mark := position of C in ParmString where(C = '/');
    ClientName := every of C in ParmString where(position of C < Mark);
    Arguments := every of C in ParmString where(position of C > Mark);
    Client := create of ParmString.GetProgram(ClientName);
    call Filter(ParmString.PutLine, "TheFilter", CliToFil);
    call Client(ParmString.GetLine, CliToFil, ParmString.GetProgram, Arguments);
    on (NotFound)
        call ParmString.PutLine("compound: Syntax Error.");
    on (GetProgramInterface.NotFound)
        call ParmString.PutLine("compound: " | ClientName | " not found.");
    on (InterfaceMismatch)
        call ParmString.PutLine("compound: " | ClientName | " interface mismatch.");
end block;
return ParmString;
```

The program begins by calling the service GetProgram, passing it the character string name of a program to be fetched from the program library. The result returned by GetProgram is a program. A program is a value of the predefined Hermes data type program—a set of declarations and statements of a process module stored as structured values rather than as source text. There are three ways to obtain values of type program: (1) by passing the source text of a Hermes process through the compiler, saving the result in a program library, and retrieving it with GetProgram, (2) by writing a "program literal", and (3) by directly building a program value "on-the-fly". In this case, we are using the first approach. We have assumed that the program named "filter" has been previously compiled and that the compiled program has been placed in a library managed by the GetProgram service.

(Note: Strongly typed languages, like Algol, Ada, Pascal, C, etc., do not support on-the-fly program creation. Some weakly typed languages

---

2 The terms strongly-typed and weakly-typed denote respectively compile-time and run-time checking for type violations.
like Lisp, Scheme, and APL, allow programs as first-class values. Hermes combines the flexibility of these languages with the static checking of the Algol-like languages. Hermes programs, even when dynamically built, are fully checked for type and typestate errors before being instantiated as processes ([SYB87a, SYB87b, SY86]).

To create a new process, you apply the operation \texttt{create of} to a value of type \texttt{program}. (Note: the syntax \texttt{create of expression} is used rather than \texttt{create(expression)} to distinguish primitive operations from function calls. Remember that function names like \texttt{Parms.GetProgram} are variables, whereas \texttt{create} is not.) The \texttt{create} operation instantiates a new (child) process. The code of the child process is given by the value of the operand to \texttt{create}. In this case, that will be the \texttt{Filter} program, obtained from the program library by \texttt{GetProgram}. The result of \texttt{create} is an output port bound to the initialization port of the filter process.

We create the client process in the same way: first we call \texttt{GetProgram} to obtain the program from the library, and then we apply the operation \texttt{create of} to this program. The only difference is that this time, the program name is not a constant, but must be extracted from \texttt{Parms.ParmString}. We will not explain this extraction in detail — one statement locates the delimiter '/', a second obtains all characters to the left of the delimiter, the third obtains all characters to the right of the delimiter. The operations \texttt{position} of and \texttt{every} of are explained in detail in section 2.6. Both processes — client and filter — have now been created.

The newly created processes cannot do anything yet. They need to be initialized, and they depend for their initialization on receiving a call from their parent. Remember that these programs (and probably most of the programs you'll ever write) begin with \texttt{receive Parms from Init}. So we now call the initialization ports of the two child processes.

For \texttt{Filter}, we need to supply the \texttt{PutLine} output port, and a prefix string. We will receive back an output port connected to the filter's input port. We store this port as a variable \texttt{CliToFil}. In the \texttt{call Client} statement, we need to supply 3 output ports, and a string. For two of the output ports — \texttt{GetLine} and \texttt{GetProgram}, we supply the ports which we ourselves received. But for \texttt{PutLine}, we don't want to give the client access to the same service which we can access. Instead, we want to give the client the port \texttt{CliToFil} — the output port we just obtained from Filter, and which is connected to Filter's input port. We also have to give the client a parameter string, so for now, let's just send an empty string. After making both calls, we are finished — the two child processes will now do all the real work.

We have written three exception handler clauses. The first clause handles a \texttt{NotFound} exception which can be raised by the operation \texttt{position} of in the event there is no delimiter. The second clause handles an exception anticipated by the \texttt{GetProgram} interface — namely that the requested program is not found in the library. The third clause handles the built-in ex-
InterfaceMismatch, which can be raised by the create statement. InterfaceMismatch is one of the few type-checking errors that cannot be detected at compile-time, but only at run-time. For example, the second create statement instantiates a child process, and returns an output port bound to the initialization port. It is known statically (that is, at compile time) that the type of this output port must be compatible with the output port Client. But it is not known statically whether the initialization input port of the program is of a matching type, since GetProgram may return an arbitrary program. If this program doesn’t have the proper S-plug interface (as, for example, Filter does not), then the InterfaceMismatch exception will be raised.

Because modules aren’t hard-wired to specific services, they can be reused. Any type of call can be rebound provided it is to a service of matching type. Think of the flexibility of UNIX pipes extended to any kind of calls.

Although this example is very simple, it illustrates the elegance of basing systems programming on dynamic process creation and dynamic connection of ports. The technique is very flexible. Programs like Compound can contain arbitrary policies for binding the ports of the programs it creates, thereby controlling their access to other processes. You don’t need to be a kernel hacker or a superuser to write access control policies. You cannot accidentally or intentionally circumvent someone else’s access control. You cannot access another user’s global variables, since there are no global variables. You cannot access a service by guessing its name, or by using some undefined value as a name. You can only get a binding to an input port P if (1) you own P yourself and connect to it, (2) P is the initialization port of a process you create, or (3) someone else gives you a binding to P.

This approach to system configuration is called capability-based. It is an old idea in operating systems. It is not widespread, because its premise—that capabilities (access rights) are easy to store and pass around, but impossible to forge—is often difficult and expensive to implement. Hermes solves this problem inexpensively by integrating the operating system primitives into the language. Type-checking guarantees that you can’t copy an integer, string, or other inappropriate datatype into a capability (output port) variable. Type-checking guarantees that you can’t use an uninitialized and possibly garbage value of a capability. But you may create capabilities with connect, and copy them or pass them around in messages as much as you please.

Summary: We have studied the program Filter—our first example of a program which was both a client and a server. You have learned that a server is typically divided into an initialization part which imports and exports ports, and a service loop which handles requests from its clients. You have learned how to implement the basic functions of a shell—creating processes, and setting up port connections between processes. You have learned that Hermes is a capability-based operating system encapsulated within a programming language. You have learned the following Hermes
constructs: the new input-port, connect and create statements, and the InterfaceMismatch exception. Congratulations! You now know six of the ten Hermes system programming statements!

1.5 Declarations and Definitions

Let’s take a break from learning new kinds of Hermes statements, and tie up a very important loose end mentioned earlier—declarations and definitions.

There is much debate and difference of opinion about the role of type checking. Should type checking be done at compile-time or run-time? How much should the programmer have to write out as explicit declarations, and how much should be inferred by the compiler? Should types be polymorphic?

Hermes takes the following positions:

- All programs will produce semantically well-defined localized effects—either a normal result or an exception. This means programming bugs can’t crash the system, or write random garbage into another process. In fact, a process can access only the data belonging to it.
- Programming errors will be detected at compile time wherever possible.
- Programmers must declare the types of all variables except expression temporaries. The compiler infers the types of expression temporaries.
- There are no polymorphic types. They’re not needed as much when you have the ability to create program objects on the fly. Instead of writing a polymorphic sort program which can sort any type, one instead writes a template and then creates specific sort programs by substituting specific types into the template. This accomplishes the same goal—namely, reusable source code packages—without explicitly introducing polymorphism. (Hermes has a polymorph type, which allows a variable to hold a value of any type, thereby deferring type checking until run-time.)

1.5.1 DECLARATIONS

We will now show you the complete text of the programs whose executable statements we showed you earlier.

```
HelloWorld: using(Standard) process(Init: StandardIn)
    declare
        Parms: StandardInterface; -- callmessage received from parent
    begin
        receive Parms from Init;
```
call Parms.PutLine("Hello, World!");
return Parms;
end process

The statements Init: StandardIn and Parms: StandardInterface are called declarations. A declaration consists of an identifier, a colon, and a type name. In HelloWorld, we declare two identifiers—Init (the initialization port), and Parms (the callmessage). A type name is either a predefined type name (such as integer, charstring, program, etc.), or a user-defined type name.

Some types, like Charstring are predefined. Others, like StandardIn are user-defined. You may group collections of definitions into files called definitions modules. You must import the definitions modules containing the definitions of the types you name in your source program. You must compile a definition module and place it into a library before you can import the module.

You import a definitions module by including its name on an imports list introduced by the word using. The types defined in the imported modules, and the predefined types are said to be visible within the source program. No other types are visible—that is, the compiler will not understand any other type names. If a type name is defined more than once (e.g. in two different imported modules, or as a predefined type and in an imported module), that name must be disambiguated by prefixing it with the name of the definitions module or with predefined.

In the example, the definitions module named Standard is imported. It will contain definitions for StandardIn and StandardInterface. The type name StandardInterface could optionally have been written as Standard! StandardInterface, and this would have been necessary if StandardInterface had also been the name of a predefined type or if another imported definitions module contained a type named StandardInterface.

The declaration of the initialization port appears in a special place—in parentheses after the word process. That's because the initialization port type is part of the interface to the process — it needs to be known by any process wishing to instantiate this process. The other declarations are internal, and appear between the keywords declare and begin.

Here is the entire program Echo:

Echo: using(Standard) process(Init: StandardIn)
declare
   Parms: StandardInterface; -- callmessage received from parent
begin
   receive Parms from Init;
   block
      declare
         Line: Charstring; -- line read from standard input
begin
while ('true') repeat
    callParms.GetLine(Line);
    callParms.PutLine(Line);
end while;
on (GetLineInterface.EndStream)
end block;
returnParms;
end process

Notice that except for the program name, the first five lines and the last two are identical to HelloWorld. It would be easy for an editor or other tool to insert these lines automatically as boilerplate around any standard client program. If this is done, then HelloWorld once again becomes a one-line program, and Hermes doesn't appear so verbose.

In this example, there is an inner block statement containing a declaration. If a variable name is declared within a block, then that name is visible only within the block. This rule is found in every block structured language. Hermes doesn't allow you to hide an outer declaration by declaring a new variable with the same name in an inner block. This avoids having "holes" in the region of visibility of a variable. This rule does not exist in most other block structured languages. You may have to watch out for name conflicts if you copy code containing blocks from one program to another.

Here are the programs Filter and Compound. This time, we'll just show the header and declarations without repeating the code:

Filter: using(Standard, Filter) process(Init: FilterIn)
declare
   Parms: FilterInterface; -- initialization callmessage
   In: PutLineIn; -- input port: requests from client
   Out: PutLineOut; -- output port: bound to putline service
   PutLineCM: PutLineInterface; -- callmessage containing the line
   Prefix: Charstring; -- string to prepend to each output line
begin
    -- code for process Filter
end process

Compound: using(Standard, Filter) process(Init: StandardIn)
declare
   Parms: StandardInterface; -- initialization callmessage
   Filter: FilterOut; -- port to initialize filter
   Client: StandardOut; -- port to initialize client
   CliToFil: PutLineOut; -- output port connected to filter
   ClientName: charstring; -- name of client program
   Arguments: charstring; -- arguments to client program
   Mark: integer; -- position of "/" delimiting name/parameters
begin
1. Introduction to Hermes

-- code for process compound
end process

You only need to declare the type of simple identifiers used as variable names. These identifiers are called base variables. Structured variables (e.g. records and calmessages) contain parts called components which are named using the dot notation common to many procedural languages. The type of a component like \texttt{Parms.GetLine} is inferred from the type of the base variable \texttt{Parms}.

When you write an expression, you are implicitly declaring a temporary variable to hold the result of that expression. Examples of expressions are function calls (e.g. \texttt{Parms.GetLine()}), arithmetic operations (e.g. \texttt{A+B}) and literals (e.g. \texttt{'true'}). The compiler infers the type of an expression using \textit{inference rules}. In certain cases, the type of an expression cannot be inferred, and you must precede the expression with a \textit{type specifier}. You may write a type specifier even when it is not necessary—for example, you may write \texttt{boolean \# 'true'} instead of \texttt{'true'}. In that case, the type you write down is checked against the type that is inferred.

1.5.2 Definitions

Now let's look at the definitions modules. Let's begin with Standard—the definitions describing what a standard client is passed on initialization, and the interfaces to the standard services \texttt{GetLine}, \texttt{PutLine}, and \texttt{GetProgram}:

\begin{verbatim}
Standard: using() definitions

PutLineInterface: callmessage(
    Line: Charstring)
constant(Line)
exit {full};
PutLineIn: inport of PutLineInterface {full};
PutLineOut: outport of PutLineIn;

GetLineInterface: callmessage(
    Line: Charstring)
exit {init(Line)}
exception EndStream {};
GetLineIn: inport of GetLineInterface {};
GetLineOut: outport of GetLineIn;

StandardInterface: callmessage(
    GetLine: GetLineOut,
    PutLine: PutLineOut,
    GetProgram: GetProgramOut,
    ParmString: Charstring)
constant(GetLine, PutLine, GetProgram, ParmString)
\end{verbatim}
exit {full};
StandardIn: import of StandardInterface {full};
StandardOut: output of StandardIn;

GetProgramInterface: callmessage(
  Name: Charstring,
  TheProgram: Program)
  constant(Name)
  exit {init(Name), full(TheProgram)}
  exception NotFound {init(Name)};
GetProgramIn: import of GetProgramInterface {init(Name)};
GetProgramOut: output of GetProgramIn;

end definitions

Module Standard does not import any other definitions modules. It may only refer to types defined within Standard itself, or to predefined types.

This module defines four call interfaces. In our previous analogy of processes as electrical components with plugs and sockets, interfaces definitions describe the shapes of the plugs and sockets. A call interface defines the information that can be communicated between two processes.

Most high-level languages provide a mechanism for defining the number and type of a procedure's parameters. Pascal also lets a programmer distinguish constant and var parameters; Ada lets you distinguish in, out, and inout parameters ([Ada83]). Hermes allows (and requires) you to specify even more information on a interface.

You define an interface separately from the processes that use it; interfaces must be defined in definitions modules and processes in process modules. This separation makes an interface available to any process that wants to use it. The compiler requires that a process module which uses an interface conform to the interface. The interface defines what a caller promises to send (and therefore what the receiver expects to receive); similarly it defines what a receiver promises to return (and therefore what the caller expects to receive upon return). Any caller and receiver which conform to the same interface can communicate.

Each interface typically includes three Hermes type definitions: a callmessage, an input port, and an output port.

A callmessage definition contains a declaration of the name and type of each parameter. In the calling process, you must supply a list of arguments of matching type. In the called process, you name the parameters using selected component notation, e.g.Parms.PutLine.

You may include a constants list in the callmessage definition. This is a list of those parameters whose values must not change during the call.

You must supply a list of the exceptions which may be returned by the called process.

You now need to supply the additional information which is required for typestate checking. A typestate is a set of program attributes known
Introduction to Hermes

 statically to hold at a particular program point. Like types, typestates are known statically. Like states, they vary from statement to statement. Certain typestate attributes are always tracked—such as whether a variable is initialized or uninitialized. Other attributes are only tracked when specifically defined by the programmer.

For now, let's just look at initialization typestates. When the attribute init(X) appears in a typestate this means that the variable X is initialized. When init(X) is absent, it is uninitialized. A Hermes variable is always statically known to be initialized or uninitialized; it is never conditionally initialized. You must supply the exit typestate—the typestate the parameters must have on normal (non-exception) return from the call. For example, the normal exit from the GetLine is init(Line), which means that an initialized Line will be returned. For each exception, you must supply the typestate the parameters must have if the call completes with that exception. The normal and exception typestates may be different. For example, if exception EndStream is returned, then Line will not be initialized, because init(Line) is not present.

An input port definition specifies the type of callmessage which will be sent to that input port, and the typestate that callmessage will have when it is sent to the input port (the entry typestate). For example, GetProgramIn defines an input port type which will hold callmessages of type GetProgramInterface. These callmessages have two components or parameters, named Name and TheProgram. A callmessage arriving at a port of type GetProgramIn is expected to have component Name initialized, but not component TheProgram, because the entry typestate is init(Name).

The abbreviation full is used to avoid writing out fully initialized typestates at length. For example, type program is a record with components Definitions, Modules, Main, Program, and Programs. We write full(TheProgram) in the interface instead of the much longer

\[
\text{init(TheProgram), init(TheProgram.Definitions, Modules), init(TheProgram.Main, Program), init(TheProgram.Programs)}
\]

Where full appears all by itself, it means that the entire callmessage is fully initialized. So the exit typestate of StandardInterface could have been written as

\[
\text{init(GetLine), init(PutLine), init(GetProgram), init(ParmString)}
\]

There are other typestate attributes besides init. For example, the attribute checked applies to a program value and it means that the program is not only fully initialized, but also that it is free of compile-time errors.Unchecked programs must be checked before they may be instantiated with

\[^3\text{You can use a variant type, described in section 4.3, to get the effect of conditional initialization.}\]
the `create` of statement. The definition `GetProgramInterface` specifies that programs returned by `GetProgram` have already been checked.

An output port definition simply specifies the type of the input port to which the output port connects—called the matching input port type. The `callmessage` type and `typestate` required for sending on the output port are determined from that input port type.

The interface definitions make `typestate` checking possible even though calling and called programs are separately compiled. When compiling the caller, the compiler checks that the entry `typestate` holds before the call, and computes the exit and exception `typestates` using the interface. When compiling the called program, the compiler uses the interface to deduce the entry `typestate`, and checks that the correct exit `typestate` holds at the point of a `return` statement.

See if you can work out for yourself what the definitions module `Filter` must look like. Then check yourself against the definition given:

```plaintext
Filter: using(Standard) definitions

FilterInterface: callmessage(
    PutLine: PutLineOut,
    Prefix: Charstring,
    ClientPutLine: PutLineOut)
constant(PutLine, Prefix)
exit {full};
FilterIn: import of FilterInterface {init(PutLine), init(Prefix)};
FilterOut: outport of FilterIn;

end definitions
```

Notice that `Filter` has to import `Standard`, since it uses a type `PutLineOut` defined in `Standard`. `Standard` refers only to predefined types and types defined within itself.

Note: Hermes uses name equivalence for type definitions. This means, for example, that if you define two different `callmessage` types which happen to have the same structure—same component types, same exceptions, etc.—they are treated as two different types. If you intend two variables to be used in the same contexts, be sure to declare them as the same type.

**Summary**: In Hermes, all base variables must be declared. All user-defined types must be defined in separate modules. Only the names of predefined types and of types defined in imported definitions modules are visible. You have learned how to write process headers, declarations, and definitions modules, and the syntax for type definitions of input ports, output ports, and `callmessages`. You know when you need to put a type specifier on an expression. You have learned the following terminology: type specifier, definitions module, import, visible, predefined, entry `typestate`, exit `typestate`, `checked` attribute, name equivalence. You now understand the entire skeleton of the Hermes language and can write and execute Her-
mes programs. The only thing you do not know now are some additional Hermes datatypes and Hermes statements.

1.6 A Simple Server

We have seen one example of a server, namely the Filter. This wasn't a typical server, since it made only a single operation available, and didn't really keep any local state, but instead passed all requests through to another service.

Let's look at a more typical server—one more like the objects of object-oriented languages. Our server will implement a buffer. A buffer supports the operations GetLine and PutLine. Calls to PutLine insert data into a first-in, first-out queue; calls to GetLine remove the data from the queue. If the queue is empty, calls to GetLine will block until the queue becomes non-empty. For a bounded buffer, calls to PutLine will block if the number of lines in the queue exceeds some capacity. We will supply one additional operation, Quit which is simply a command to terminate. Figure 1.6 shows the interface.

Naturally, we implement a bounded buffer in Hermes with a process. Here is the source module:

BoundedBuffer: using(Standard, BBExternal, BBLocal, Quit) process(Init: BBIn)
declare
Parms: BBInterface; -- initialization callmessage
Put: PutLineIn; -- PutLine service
Get: GetLineIn; -- GetLine service
Quit: QuitIn; -- Quit service
Capacity: integer; -- maximum number of messages
Queue: Lines; -- ordered set of lines
Running: boolean; -- true if process has not been shut down
GetCM: GetLineInterface;
PutCM: PutLineInterface;
QuitCM: QuitInterface;
begin
  -- initialization
  receive Parms from Init;
new Put; connect Parms.Put to Put;
new Get; connect Parms.Get to Get;
new Quit; connect Parms.Quit to Quit;
Capacity := Parms.Capacity;
new Queue;
return Parms;
  -- service loop
  Running := 'true';
  while (Running) repeat
    select
      event Put and where(size of Queue < Capacity)
        receive PutCM from Put;
        insert copy of PutCM.Line into Queue;
        return PutCM;
      event Get and where(size of Queue > 0)
        receive GetCM from Get;
        remove GetCM.Line from Queue[0];
        return GetCM;
      event Quit
        receive QuitCM from Quit;
        Running := 'false';
        return QuitCM;
      otherwise -- should not occur
        end select;
  end while;
end process

The BoundedBuffer process supports three operations. It supplies an input port for each operation. The parent will call the initialization port and pass a parameter Capacity. On return, it will receive connections to the three input ports. The initialization code saves the Capacity parameter, creates the three input ports, makes connections to these input ports, and returns.

The service loop consists of a construct called a guarded select statement. The statement contains clauses called alternatives. Each alternative has an optional event guard, naming an input port, and an optional boolean guard. When the select statement is executed, all the boolean guards are evaluated. Alternatives with a false boolean guard are disabled; the alternatives with a true boolean guard or no boolean guard are enabled.
If all alternatives are disabled, then the otherwise clause is executed. If one or more alternatives are enabled, one of them is chosen, and control transfers to that alternative. However, an alternative with an event guard can only be chosen if that input port has at least one waiting callmessage. If all the enabled alternatives have event guards referring to empty input ports, the select statement blocks until a message arrives at one of these ports.

In this example, the Quit alternative is always enabled; the Put alternative is enabled provided the number of lines in the queue is less than Capacity, and the Get alternative is enabled provided there are any lines in the queue. The otherwise clause, although syntactically mandatory, can never be reached. It is easy to modify this program into an unbounded buffer. Just leave out the queue size test from the Put service.

The bounded buffer process uses four interfaces: (1) BBLocal (not shown), which contains definitions for types used internally by BoundedBuffer (in this case, the type Lines used to implement the queue), (2) BBExternal, which defines the initialization interface for the bounded buffer, (3) Standard, which defines the PutLine and GetLine, and (4) Quit, which defines the interface for the Quit operation. Standard and Quit are in separate modules, because they will be reused in applications other than the bounded buffer.

The interface definition, BBExternal, is straightforward, and uses only language features we've already described. Try to write it yourself and then check with the definition here:

```
BBExternal: using(Qui1t, Standard) definitions
BBInterface: callmessage(
  Capacity: integer, -- maximum number of messages allowed
  Put: PutLineOut, -- port for writers
  Get: GetLineOut, -- port for readers
  Quit: QuitOut) -- port for shutdown
constant(Capacity)
exit {full};
BBIn: inport of BBInterface {init(Capacity)};
BBOut: outport of BBIn;
end definitions
```

The Quit definition is even simpler:

```
Quit: using() definitions
  QuitInterface: callmessage() exit {};  
  QuitIn: import of QuitInterface {};  
  QuitOut: outport of QuitIn;
end definitions
```

How does Hermes differ from data abstraction languages like Ada or object-oriented languages like Smalltalk? In both cases, we have the ability
to write data abstractions which export a specific set of operations while not exposing the implementation details to the caller. There are some differences:

- Hermes processes are active. If there are multiple callers, the calls are queued and processed serially. The process can choose not to receive certain calls at certain times. There is no need for monitors or semaphores or other mutual exclusion mechanisms.

- I can write different Hermes processes with the same interface and different internals. In Ada, for instance, I can't have two variables of the same private type with the same specification (interface) and different private parts.

- I control access in Hermes on a port-by-port basis. In Ada or in object-oriented languages, I control access on an object basis. Having access to an object automatically gives access to all its operations. In the case of a bounded buffer, this would mean that anyone who could call GetLine could also call PutLine and Quit. This is undesirable, and in fact I don't want the readers and writers of the bounded buffer to even have to know that the Quit operation exists.

The Queue is an example of a type family we haven't studied before—an ordered table. A table is simply a collection of values of the same type (mathematically, a bag). An ordered table is a table whose elements are totally ordered—that is, each element has an associated position. Positions are denoted by integers. The first position is 0. If there are \( k \) elements in the table, the last position is \( k - 1 \).

You have already used ordered tables without knowing it, since the predefined type Charstring is an ordered table of elements of type Char. In Hermes you can define tables of any type—tables of integers, tables of callmessages, etc. You can even define a recursive table type—a type \( T \) which is a table whose elements are of type \( T \). In our example, we want to define a type Lines as an ordered table whose elements are of type Charstring. The type definition looks like this and will appear in the definitions module BBLocal:

```plaintext
Lines: ordered table of Charstring {init};
```

The definition says that Lines is an ordered table, that each element is a Charstring, and that elements inserted into, removed from, or copied from the table have typestate init. Notice that like other type definitions in Hermes, and unlike type definitions in many other languages, table type definitions say nothing about the representation (e.g., array, linked-list, disk file), and nothing about how much storage to allocate.

All tables support the following operations: inspect an element in the table with a particular content, insert elements into the table, remove elements from the table, merge one table into another, extract or copy out a
subsetable from a table, find the size of the table, and iterate (repeat a set of statements once for each element of the table).

Ordered tables additionally support these operations: insert, remove, or inspect an element or merge a table at a particular position, locate the position of an element with particular contents, and iterate in order.

In BoundedBuffer, the single action of the Put service consists of inserting the designated line into the end of the table Queue. The insert statement inserts an element into the end of a table by default if the table is ordered. However, you may insert elements at any position in the table if you wish, by writing:

\begin{verbatim}
insert Element into Table at (Position);
\end{verbatim}

In this case, you must supply an integer-valued position. For example, if you specify position 3, the new element appears at position 3. If there was an element previously at position 3, it moves to position 4, and similarly all elements at higher positions have their positions incremented. If you try to insert something at position 3, there must be an element at position 2, or else you will get a \texttt{RangeError} exception.

One more technical point about \texttt{insert}: the statement \texttt{moves} its operand into the table, rather than copying it. This means that after the statement is executed, the source operand is discarded. Since we're not allowed to discard \texttt{PutCM.Line}—in the definition of \texttt{PutLineInterface}, parameter \texttt{Line} is constant—we must move a copy of \texttt{PutCM.Line}. That explains why we wrote the expression \texttt{copy of}.

The \texttt{Get} service clause removes the first entry from the table and places it in \texttt{GetCM.Line}. When you remove the first element from an ordered table, the positions of all the other elements move up by one, so the previous second element now becomes first.

The \texttt{Quit} service call terminates the loop, and hence terminates the process.

Let's look more closely at process termination. A process terminates after executing its last statement. All variables which remain initialized are finalized—that is, their values are discarded. In BoundedBuffer, that would be the queue, the boolean variable \texttt{Running}, the three service ports, and the initialization port. For most Hermes data types, discarding the value means simply throwing it away. Callmessages and ports are treated specially: When a callmessage, or any value which can contain callmessages (e.g., an input port which may have several callmessages on its queue) is discarded, the callmessages are returned. Each callmessage type has a default \texttt{Discarded} exception, e.g., \texttt{GetLineInterface.Discarded}. When an input port is discarded, connections to that port are broken; attempts to send messages over those connections will fail and the caller will receive a \texttt{Disconnected} exception. Any process which becomes isolated because all its connections to the outside world are broken is equivalent to a terminated process and can be safely garbage-collected by the implementation.
This has the following consequences: To give a process temporary access to a service, while retaining the ability to revoke access, give out a connection to a filter process which also supports a Quit service that you call when you want to revoke access. If you're programming a shell which is going to create processes running arbitrary programs which you're afraid might run wild, make sure that on initialization, you only pass revocable output ports. Then if you revoke all the ports, the process will become isolated, effectively cancelling the process.

**Summary:** We have shown the structure of a typical server: an input port per service, some local state, and an iterated `select` statement. Boolean guards may be used to prioritize the calls or to conditionally disable certain services. We have claimed that server processes achieve the information hiding of data abstraction and object-oriented languages but with additional advantages. You have learned about the `table` and `ordered table` type families. You have learned what happens when processes terminate, and that you can use the ability to terminate filters as a means for access revocation and termination of child processes. You have learned the following Hermes concepts: alternative clauses, enable and disable, boolean guard, event guard, Discarded exception, Disconnected exception, RangeError exception, process termination, the `select`, `insert` and `remove` statements, the copy of and size of operations.
A Miniature System

Our final example will be a dynamic system—a simplified window system. Since the real purpose of this section is to illustrate systems programming with Hermes and not to design the ideal window system, we will eliminate the hairy details of graphics, tiling, etc., and strip the problem to the bare bones.

Most of the Hermes constructs used in this example are already familiar; however we will introduce and explain a few new constructs as we go along.

2.1 Requirements

Suppose I have a terminal device. Let’s assume that it uses the same oversimplified terminal interface we have been using in the previous examples: GetLine and PutLine. I now would like to multiplex this terminal so that I can run several client applications. Each client application was originally written to use the terminal device, so they use the GetLine and PutLine interfaces, too.

I want to divide my terminal into logical windows. Each application runs in its own window. Output from an application will be directed to its window. Input from the terminal will be directed to the window currently in focus. From the terminal, I can also (1) change the current focus, (2) start up a new application in a new window, or (3) kill a window together with the application running in it.

In this example, we will implement these functions, using our limited, line-at-a-time interface, as follows: Every window will have a character string name. When a line is written out by an application, it will appear on the terminal in the form `<WindowName>:<Line>`. To remind the user which window is in focus, the name of window currently in focus will be typed out as a prompt. Every input line not headed by an escape character (!) is assumed to be typed into the current window. Every input line headed by an escape character is treated specially by the window manager. There are five escape sequences:

- `!!line` - dispatch `!line` to the window currently in focus. This is how you send data which begins with an escape character.
- `!C windowname application parms` - create a window `windowname` running `application`
- `!F windowname` - change focus to `windowname`
2.1. Requirements

This is obviously not a realistic window manager, but its structure can be used as a guide to building something more practical. It wouldn’t be hard to modify the code to use a graphical screen, and to use mouse clicks rather than escape sequences to change the focus and to create and destroy windows. The basic structure of the system remains the same.

2.2 Design

The structure of the system is shown in figure 2.1.
2. A Miniature System

A front-end process reads lines from the terminal, parses them, and invokes one of the services: Refocus, Kill, Create, or Dispatch. The front-end also keeps track of which window (if any) is currently in focus.

The window manager process supports these four services called by the front end. The window manager process services WriteToWindow calls from applications sending lines to their window. For each window, the window manager has a pair of ports: (1) InputToWindow, for directing data to the application running in that window, and (2) Quit to shut down the window.

Each application consists of three processes: (1) the client itself, with the same standard interface familiar from HelloWorld, (2) an unbounded buffer process, similar to the BoundedBuffer described above which stores lines typed ahead to the client, and (3) an adapter process, Adapter. These processes are shown in detail in figure 2.2. The adapter is similar to Filter. It converts PutLine calls into the appropriate WriteToWindow calls, and passes GetProgram calls. Both the unbounded buffer and the adapter are prepared to receive Quit calls when the window manager decides to kill the window. When a Quit call is received, the unbounded buffer and the adapter shut down, isolating the client and thus terminating it.

2.3 Interfaces

Once we have designed the module structure, we can write the definitions modules. Here are the interface definitions for the window manager:

\[
\text{WMExternal: using(Standard) definitions}
\]
2.3. Interfaces

-- initialization interface for window manager
WMInterface: callmessage (  
  GetProgram: GetProgramOut, -- service for loading programs
  PutLine: PutLineOut, -- write a line to the physical terminal
  Dispatch: DispatchOut, -- dispatch a line to the appropriate window
  Refocus: WindowOut, -- change the current focus
  Kill: WindowOut, -- kill the specified window
  Create: CreateOut) -- make a new window
  constant(GetProgram, PutLine)
  exit {full};
WMIn: inport of WMInterface {init(GetProgram), init(PutLine)};
WMOut: outport of WMIn;

-- interface to operation Dispatch
DispatchInterface: callmessage (  
  WindowName: Charstring, --- window to which text is being sent
  Line: Charstring) -- text to dispatch
  constant(Line)
  exit {full}
  exception NotFound {full}; -- no such window
DispatchIn: inport of DispatchInterface {full};
DispatchOut: outport of DispatchIn;

-- interface to operations Refocus, Kill
WindowInterface: callmessage (  
  WindowName: Charstring) -- name of window operand
  constant(WindowName)
  exit {full}
  exception NotFound {full};
WindowIn: inport of WindowInterface {full};
WindowOut: outport of WindowIn;

-- interface to operation Create
CreateInterface: callmessage (  
  WindowName: Charstring, -- name of window being created
  ProgramName: Charstring, -- name of client program to run
  ParmString: Charstring) -- parameters to client program
  constant(WindowName, ProgramName, ParmString)
  exit {full}
  exception Duplicate {full}
  exception CreateFailure {full};
CreateIn: inport of CreateInterface {full};
CreateOut: outport of CreateIn;

-- interface to operation WriteToWindow
WriteToWindowInterface: callmessage (  
  WindowName: Charstring, -- name of window to which text is being sent
  Line: Charstring) -- text to dispatch
  constant(Line)
  exit {full}
  exception NotFound {full};
WriteToWindowIn: inport of WriteToWindowInterface {full};
WriteToWindowOut: outport of WriteToWindowIn;
Each window application needs to be initialized with a program name (needed by the application builder) and parameters (needed by the client), access to the WriteToWindow and GetProgram services, and the name of the window (needed by the filter). The application must export connections to the buffer process and to the quit service. So here are the definitions for the initialization of an application:

```
StartWindowApplication: using(Standard, BBExternal, WMExternal, Quit)
definitions

WindowApplicationInterface: callmessage(
    ProgramName: Charstring, -- name of client program to run
    ParmString: Charstring, -- parameters to client program
    GetProgram: GetProgramOut, -- service for loading programs
    WriteToWindow: WriteToWindowOut, -- for writing lines to a
    WindowName: Charstring, -- name of window
    InputToWindow: PutLineOut, -- for writing into input buffer
    Quit: QuitOut) -- for killing the application
constant(ProgramName, ParmString, GetProgram, WriteToWindow,
    WindowName)
exit {full}
exception NotCreated
{init(ProgramName), init(ParmString),
    init(GetProgram), init(WriteToWindow), init(WindowName)};

WindowApplicationIn: inport of WindowApplicationInterface
{init(ProgramName), init(ParmString),
    init(GetProgram), init(WriteToWindow), init(WindowName)};

WindowApplicationOut: outport of WindowApplicationIn;

definitions

The three processes comprising an application are: the client, the bounded buffer, and the adapter. The client will have interface Standard. This follows from the original problem statement which specified that any program which had been written to communicate with the terminal can run within a single window. The bounded buffer has interface BBExternal, and the adapter has interface Adapter. We have already designed BBExternal. Since the adapter is like the filter, we will design its interface by making
a small change to the interface Filter. Recall that the adapter exports a port to its Quit service as well as a ClientPutLine port:

```
Adapter: using(Standard, WMExternal, Quit) definitions
```

```
AdapterInterface: callmessage(
    AdapterToWindow: WriteToWindowOut, -- port to window manager
    WindowName: Charstring, -- window name for directing output
    GetProgram: GetProgramOut, -- port to GetProgramService
    Quit: QuitOut, -- port to shutdown adapter
    ClientPutLine: PutLineOut, -- port from client to adapter
    ClientGetProgram: GetProgramOut) -- port from client to adapter
constant(AdapterToWindow, WindowName, GetProgram)
exit {full};
```

```
AdapterIn: import of AdapterInterface
    {init(AdapterToWindow), init(WindowName), init(GetProgram)};
```

```
AdapterOut: outport of AdapterIn;
```

```
end definitions
```

```
Finally, the front end process must be given the GetLine, PutLine, and GetProgram ports, and the ports to the four front-end accessible services of the window manager. Its interface is:

```
FEPExternal: using(Standard, WMExternal) definitions
```

```
FEPInterface: callmessage(
    GetLine: GetLineOut,
    PutLine: PutLineOut,
    GetProgram: GetProgramOut,
    Dispatch: DispatchOut, -- dispatch a line to the appropriate window
    Refocus: WindowOut,
    Kill: WindowOut,
    Create: CreateOut)
constant(GetLine, PutLine, GetProgram, Dispatch, Refocus, Kill, Create)
exit {full};
```

```
FEPIn: import of FEPInterface {full};
```

```
FEPOut: outport of FEPIn;
```

```
end definitions
```

### 2.4 Window System Shell

Now that the interfaces are all written, the programs are fairly easy to write. Let's begin with the outer shell of the system. It simply creates the front end process and the window manager. The outer shell is similar to the program
Compound which we wrote earlier. It creates the window manager process. It calls the initialization port of the window manager, giving it access to `GetProgram` and `PutLine`, whereupon it receives back ports to four of the five window manager services. (The fifth service, `WriteToWindow`, is only available to created applications and none have been created yet.) It then initializes the front end process, passing `GetLine`, `GetProgram`, and the four services:

```plaintext
WMSystem: using(Standard, WMExternal, FEPExternal)
process(Init: StandardIn)
declare
  Parms: StandardInterface; -- initialization callmessage
  Dispatch: DispatchOut; -- dispatch service in WM
  Refocus: WindowOut; -- refocus service in WM
  Kill: WindowOut; -- kill service in WM
  Create: CreateOut; -- create service in WM
begin
  receive Parms from Init;
  call (WMOut # (create of Parms.GetProgram("windowmanager")))
    (Parms.GetProgram, Parms.PutLine, Dispatch, Refocus, Kill, Create);
  call (FEPOut # (create of Parms.GetProgram("frontend")))
    (Parms.GetLine, Parms.PutLine, Parms.GetProgram, Dispatch, Refocus, Kill, Create);
  return Parms;
end process
```

### 2.5 Front-end Process

The front-end process is a simple loop in which a line is read, and a decision tree is followed to one of the decisions `Dispatch`, `Refocus`, `Kill`, `Create`, or error. Here is the program. We leave it to you to fill in the declarations.

```plaintext
running := 'true';
receive Parms from Init;
tokenizer := procedure of Parms.GetProgram("tokenizer");
WindowName := ""; -- no window in focus
block begin
  while (running) repeat
    block declare
      Line: Charstring; -- line read by frontend
      Escape: Char; -- escape character removed from line
    begin
      call Parms.PutLine( WindowName | "">
```
call Parms.GetLine(Line);
if size of Line = 0
then
  call Parms.Dispatch(WindowName, Line);
else if Line[0] <> '!'
then
  call Parms.Dispatch(WindowName, Line);
else
  remove Escape from Line[0];
block declare
  Cmd: Charstring; -- command name following !
begin
  if Line[0] = '!' 
  then
    call Parms.Dispatch(WindowName, Line);
  else
    extract cmd from Line[0];
    select (cmd)
where("F")
  block declare
    FParm: Charstring; -- parameter to !F
begin
  FParm := tokenizer(Line);
  call Parms.Refocus(FParm);
  call Parms.PutLine("!wfe: Focus changed to " | FParm | ",
                  WindowName := FParm;
  on (WindowInterface.NotFound)
    call Parms.PutLine("!wfe: " | FParm | " not a window.");
end block;
where("K")
  block declare
    KParm: Charstring; -- parameter to !K
begin
  KParm := tokenizer(Line);
  call Parms.Kill(KParm);
  call Parms.PutLine("!wfe: " | KParm | " killed.");
  on (WindowInterface.NotFound)
    call Parms.PutLine("!wfe: " | KParm | " not a window.");
end block;
where("Q")
  running := 'false';
call Parms.PutLine("Quitting window manager.");

where("C")
block declare
  CParm: Charstring; -- window parameter to
  PName: Charstring; -- program name
  Parm: Charstring; -- program parameters
begin
  CParm := tokenizer(Line);
  PName := tokenizer(Line);
  block begin
    Parm := tokenizer(Line);
    on (TokenizeInterface.noToken)
      Parm := "";
    end block;
    call Parms.Create(Cparm, Pname, Parm);
    WindowName := CParm;
    on (CreateInterface.Duplicate)
      call Parms.PutLine("wfe: Cannot window create " | CParm | 
        " since it already exists." );
    on (CreateInterface.CreateFailure)
      call Parms.PutLine("wfe: Unable to create application process " 
        |PName| 
        ");
    end block;
  otherwise
    -- error: not F, Q, K or C
    call Parms.PutLine("wfe: Illegal command: " | cmd);
  end select;
end if;

on (TokenizeInterface.noToken)
  -- error
  call Parms.PutLine("wfe: Missing argument.");
end block;
end if; end if;

on (DispatchInterface.NotFound)
  call Parms.PutLine("wfe: Window " | WindowName | " does not exist");
end block;
end while;
on (others)
call Parms.PutLine(""wfe: others exception");
end block;
return Parms;

The select statement in the example is the usual abbreviation for comparing a value against a set of values. It is equivalent to:

\[
\text{Temp} := \text{Line}[\text{ColonPosition} + 1];
\]

\[
\text{select}
\begin{align*}
\text{where}(\text{Temp} = 'F') & . . . \\
\text{where}(\text{Temp} = 'K') & . . . \\
\text{where}(\text{Temp} = 'C') & . . . \\
\text{otherwise} . . .
\end{align*}
\text{end select;}
\]

Notice that we have separated the code which parses the command from the code which obeys the commands. This style of writing has several advantages: (1) It is easy to use the commands with an alternative front-end, e.g., one based upon pointing, or dialog, or menu-selection. This exploits Hermes' ability to plug and unplug modules in different configurations. (2) There are tools which automatically generate programs such as the above front end from a simple description of the command syntax.

The front-end process uses a procedure named Tokenizer to extract a token (string of contiguous non-blank characters) from a string. Procedures fit very nicely into the process paradigm of Hermes. A procedure is simply a process which is created just for a single call, and which terminates right after it returns.

Since procedures are so common, Hermes provides a special operation, procedure of, to make it easy to invoke procedures. Without it, you would have to issue a create operation before each procedure call.

Here is how procedure of works: You code procedure of with a program operand, just like create of. The result is an output port connected to an initialization port. However, each time you make a call on that output port, a new process is instantiated, and your call is given to the initialization port of the new process.

If you are not using a procedure re-entranently or recursively — that is, there exists only one activation at a time, then it doesn't matter whether you create a regular process with create or a “process-generator” with procedure. But if you ever make a second call while the first call is still being processed, then it does make a difference: if your procedure was implemented with create of, the second call will block; if you used procedure of, the second call will not block because a new instance will be created automatically.

Processes created automatically with procedure of are still like Hermes processes in every other way. In particular, they still persist after a return statement, unless the return is the last statement. If you want the equivalent of Algol procedures, be sure to use procedure of and begin each
procedure process with a receive and end it with a return.

2.6 Tokenizer Procedure

Here is the code of the Tokenizer procedure:

```plaintext
receive Parms from Init;
pNB := position of C in parms.string where(C <> ' ');
extract blanks from B in parms.string where(position of B < pNB);
block begin
  pB := position of B in parms.string where(B = ' ');
on (NotFound)
    pB := size of parms.string;
end block;
extract parms.token from C in parms.string where(position of C < pB);
return Parms;
on (NotFound)
  return Parms exception NoToken;
end block;
end process
```

The program uses some operations on tables which have not yet been explained. The table type family is very important, since it is the single Hermes construct for representing collections of data values. Tables subsume strings, lists, arrays, files, trees, associative memories, etc. We have used tables in a number of examples, but have not explained table operations in detail.

Many table operations contain a construct called a selector. A selector has the form: element-variable in table where(predicate). A selector determines a subset of elements of a table where a boolean predicate is true. The predicate is applied to each element of the table.\(^1\) The element variable is automatically declared to have as its type the type of elements of the table. The declaration is visible only inside the where expression. For instance, in C in parms.string where(C = ' '), Parms.String is a Charstring, so C is a Char.

For ordered tables, you may code the operator position of element-variable within the predicate. The result of position of is the position of

\(^1\)This explanation, and similar descriptions of Hermes semantics, should be understood only as a definition of program behavior, not as a guide to performance complexity. Compilers are free to perform optimizations so long as an outside observer cannot distinguish the optimized and unoptimized programs. Most compilers will not literally implement x in S where(position of x = ?) by scanning each element of S, but instead will use indexing.
the element being tested by the selector. For example, *element in table where(position of element = expr)* means "select the element whose position in the table is *expr."" This form of selector works exactly like array indexing. It is so common that Hermes allows you to use the abbreviation *table[expr]*. We have encountered this abbreviation in the program BoundedBuffer. The statement *remove cmd from Line[0]* is a shorthand for *remove cmd from C in Line where(position of C = 0)*, and it means "remove the first character from Line and store it in cmd."

A selector can appear either by itself as an expression, or it can appear as part of another operator: e.g. *remove, every, extract, position*:

- A selector all by itself in an expression returns a copy of the selected element. You can look this operation up in the reference manual under the name *the-element*. If there is more than one selected element, then the earliest one is chosen if the table is ordered, and an arbitrary one is chosen if the table is unordered. A *NotFound* exception is raised if there is no selected element. Example: *Line[0]* evaluates to the first character of Line, if Line is non-empty, otherwise it raises a *NotFound* exception. It can also be written *C in Line where(position of C = 0)*.

- The *remove* statement removes a single element. If no element is selected, a *NotFound* exception is raised. If more than one element is selected, the first selected element is chosen if the table is ordered, otherwise an arbitrary choice is made.

- The *every of* operation generates a new table consisting of every selected element. If the original table was ordered, the table of selected elements will have the same relative order. This operation is used to compute subsets or substrings. (Refer back to the program Compound, where this operation was used to select those characters before and after the '/'.

- The *extract* statement is similar to *every of*, except that it removes the selected elements from its source operand, rather than copying the elements.

- The *position of* operation with a selector (e.g. *position of C in Line where(C = ' :')*) returns the position of the first selected element within the ordered table. If no element in the table is selected, a *NotFound* exception is generated.

Note the difference between *position of selector*, which searches a table, and the *position of* operator discussed above, which is applied within a selector predicate. The following expression uses both kinds of *position of* operators: *position of C in Line where(position of C > P and C = ' :')*. This expression searches for the first colon in Line after position P, and returns the position of that colon.
2.7 The Window Manager

2.7.1 Definitions

Now let's code the window manager process itself. Besides its five input ports, and its output port, what state does it need? The window manager must know which windows exist, and how to dispatch lines to those windows.

The natural Hermes way to do this is to define a table. Unlike the tables we've used so far, this table will be unordered. There will be one element in the table for each created window. The element will consist of three pieces of information: the window's name, the port over which we dispatch lines destined for the application in that window, and the Quit port which we need when we want to kill the window and isolate the client process.

We need to define two Hermes types: a record containing the window name and the two ports, and a table whose elements are records of this type. Here is what we need to write in \textit{WMInt}, the definitions module for types used internally by the window manager:

\begin{verbatim}
WMInt: using(Standard, Quit) definitions
  Window: record /
    WindowName: Charstring, -- the name of the window
    InputToWindow: PutLineOut, -- port for writing into input buffer
    Quit: QuitOut\); -- port for killing the application

  Windows: table of Window {full} keys(WindowName);
end definitions
\end{verbatim}

The record definition is straightforward. The table definition says that a table of type \textit{Windows} consists of fully initialized records of type \textit{Window}. If you are familiar with the terminology of relational databases, you will see that \textit{Window} defines a \textit{tuple} and \textit{Windows} a \textit{relation}. The notation \texttt{keys(WindowName)} says that no two elements of the table may have the same value of component \texttt{WindowName}. In relational database terminology, there is a \textit{functional dependency} between the component \texttt{WindowName} and the other components.

The semantic effect of the \texttt{keys} declaration is this: an attempt to insert a record whose \texttt{WindowName} component duplicates the \texttt{WindowName} component of a record already in the table will fail. The exception \texttt{DuplicateKey} will be raised.

There is also a syntactic effect of the \texttt{keys} declaration: the shorthand \texttt{T[N]} is available as an abbreviation for \texttt{E in T where(E.WindowName = N)}. This abbreviation exists simply because it's fairly common to search a table on its key components. If the \texttt{Windows} table type had been ordered, you would not be able to use this abbreviation, because it would instead mean \texttt{E in T where(position of E = N)}. The abbreviation for selecting
an element by position takes precedence over the abbreviation for selecting an element by key. Abbreviations are expanded before types are checked, so the fact that N is not an integer will not help.

Once again, we remind you that the physical representation of the table is hidden and in no way affects the semantics.

2.7.2 Skeleton

We're now ready to write the skeleton of the window manager:

```plaintext
WindowManager: using(Standard, WMExternal, WMInt, StartWindowApplication)
process (Init: WMIn)
declare
  Parms: WMInterface; -- initialization parameters
  Refocus: WindowIn;
  RefocusCM: WindowInterface;
  WriteToWindow: WriteToWindowIn;
  WriteToWindowCM: WriteToWindowInterface;
  Dispatch: DispatchIn;
  DispatchCM: DispatchInterface;
  Create: CreateIn;
  CreateCM: CreateInterface;
  Kill: WindowIn;
  KillCM: WindowInterface;
  GetProgram: GetProgramOut;
  PutLine: PutLineOut;
  WriteToWindowCapability: WriteToWindowOut;
  ApplicationBuilder: Program;
  CreatedWindows: Windows; -- status of all windows
  CurrentFocus: Charstring; -- name of window in focus
  CurrentWindow: Window; -- a window being insert/deleted/searched
begin
  -- initialization section
  receive Parms from Init;
  GetProgram := Parms.GetProgram;
  PutLine := Parms.PutLine;
  ApplicationBuilder := Parms.GetProgram("applicationbuilder");
  new Refocus; connect Parms.Refocus to Refocus;
  new WriteToWindow; connect WriteToWindowCapability to WriteToWindow;
  new Dispatch; connect Parms.Dispatch to Dispatch;
  new Create; connect Parms.Create to Create;
  new Kill; connect Parms.Kill to Kill;
  return Parms;
  new CreatedWindows;
  -- service section
```
while ('true') repeat
  select
    -- event Refocus ...
    -- event WriteToWindow ...
    -- event Dispatch ...
    -- event Create ...
    -- event Kill ...
    otherwise
  end select;
end while;
end process

So far, the window manager has the same structure as servers such as BoundedBuffer. One difference is that we do not give the window manager's parent a connection to the WriteToWindow service. This connection will be given instead to the children of the window manager—the window applications.

2.7.3 Refocussing and Writing Output

The Refocus and WriteToWindow services are straightforward. Refocus simply checks whether the window exists.

```
event Refocus
  receive RefocusCM from Refocus;
  if exists of CreatedWindows[RefocusCM.WindowName] then
    return RefocusCM;
  else
    -- error action
    return RefocusCM exception NotFound;
  end if;
WriteToWindow prepends the window name to the text line.
```

```
event WriteToWindow
  receive WriteToWindowCM from WriteToWindow;
  call PutLine(WriteToWindowCM.WindowName | "": " | WriteToWindowCM.Line);
  return WriteToWindowCM;
```

2.7.4 Dispatching Lines to a Particular Window

The Dispatch service involves a simple table lookup. We return an exception if the destination window does not exist.

```
event Dispatch
  block begin
    receive DispatchCM from Dispatch;
    CurrentWindow := CreatedWindows[DispatchCM.WindowName];
```
This program illustrates one way to look up something in a table—use a selector expression to copy an element from the table into a variable. We should point out one situation in which this does not work. The input port and callmessage type families are not copyable. Variables of these types, or of any other type containing these types (such as a record containing an input port or a table containing a callmessage) cannot be copied. To do a table lookup when the table elements cannot be copied, use the inspect statement. Here is the same program fragment rewritten with the inspect statement.

```haskell
block begin
  receive DispatchCM from Dispatch;
  inspect W in CreatedWindows[DispatchCM.WindowName]
    begin
      call W.InputToWindow(DispatchCM.Line);
    end inspect;
  on(NotFound)
    end block;
```

The inspect statement assigns W from a constant copy of the selected element of table CreatedWindows. Hermes does not allow making copies of input ports and callmessages, because the semantics of receive and return would become complex. However constant copies may be made of an object of any datatype, because the problematic operations are illegal when applied to constants.

In the program fragment written without inspect, it is necessary to declare CurrentWindow. But in the version using inspect, you don't write a separate declaration for the variable W. The inspect statement defines a new scope in which W is declared to have type Window—the element type of table CreatedWindows. You may wish to use the inspect statement for table lookup all the time, even when the table elements are copyable. That way, you will not have to use two different styles for the copyable and non-copyable cases.

The inspect statement selects exactly one element (or else raises an exception). To select all elements, iteratively, use a for statement. The for statement iteratively executes the statements within its body once for each element of a table which satisfies the selector predicate. (This may be zero times if the table is empty or if no element satisfies the selector.) For example, here is a piece of code which prints the names of all the active windows:

```haskell
for Window in CreatedWindows where('true')
inspect
call PutLine(Window.WindowName);
end for;

Like the inspect statement, the for statement creates a scope. Therefore you do not declare Window explicitly. Within that scope, the variable Window contains a constant copy of one of the selected elements of the table. For ordered tables, the iterations occur in the order that the selected elements occur in the table. For unordered tables, the iterations occur in an arbitrary order.

The construct where('true') may be abbreviated as []). This abbreviation is particularly useful in the for statement.

2.7.5 Creating and Killing Windows

Now let's do creation and destruction of windows. To create a window, we're going to build a new window record, and insert it into the table. The new window record contains the window's name, and the two ports InputToWindow and Quit. We get these ports by creating an application builder process. The application builder will create the three application processes, and return the two desired ports. We have to pass the application builder these parameters: the program name and parameter string, a connection to GetProgram, and a connection to WriteToWindow. The code is very straightforward now that we have all the interfaces and data structures defined:

event Create
  receive CreateCM from Create;
  if exists of CreatedWindows[CreateCM.WindowName]
    then
      -- error action because window already exists
      return CreateCM exception Duplicate;
  else
    block begin
      new CurrentWindow;
      CurrentWindow.WindowName := CreateCM.WindowName;
      call (WindowApplicationOut # (create of ApplicationBuilder))
        (CreateCM.ProgramName, CreateCM.ParmString,
         GetProgram, WriteToWindowCapability, CurrentWindow.WindowName,
         CurrentWindow.InputToWindow, CurrentWindow.Quit);
      insert CurrentWindow into CreatedWindows;
      return CreateCM;
    end block;
  on (WindowApplicationInterface.NotCreated)
    -- error action if unable to create process
    return CreateCM exception CreateFailure;
end block;
end if;

The only tricky bit is that we must avoid the possibility of a DuplicateKey exception by checking first for the existence of a window with identical name. If we waited to build the application first and then check for a DuplicateKey exception, the illegal application may have already started running and may have already produced output. The exists operator checks for this case. This operation takes a selector and evaluates to true if and only if one or more selected elements exist.

To destroy a window, we simply remove the window descriptor from the table of created windows and then call the Quit port associated with that application. This service will call Quit on the buffer process and the adapter process, with the result that the client process will shut down.

```plaintext
event Kill
receive KillCM from Kill;
block begin
  remove CurrentWindow from CreatedWindows[KillCM.WindowName];
call CurrentWindow.Quit();
discard CurrentWindow;
return KillCM;
on (NotFound)
  -- error action
  return KillCM exception NotFound;
end block;

event GetFocus
receive GetFocusCM from GetFocus;
if CurrentFocus <> ""
then
  GetFocusCM.Focus := CurrentFocus;
  return GetFocusCM;
else
  return GetFocusCM exception NotFound;
end if;
```

This finishes the window manager. Notice how convenient it is to write this code when you have tables, exception handling, and the ability to pass ports from process to process.

### 2.8 Creating a Window Application

All that is left are the application builder, the adapter, and the quit dispatcher. These programs use no new principles. If you can build Compound or WMSystem, it should be easy to build ApplicationBuilder. And Adapter is a straightforward variation of Filter.
2.8.1 Application Builder

Here is the code of ApplicationBuilder. It creates an instance of the unbounded buffer, an instance of the adapter, and an instance of the client. It also builds a process which interprets the Quit call message by calling the Quit service in both the buffer and the adapter. Here is the only tricky part: ApplicationBuilder trusts the buffer, the adapter, and the quit handler processes. It doesn't trust the client, since this might be an arbitrary module, which might not even return from its initialization call. So ApplicationBuilder should be sure to call the client last. If the client goes into a loop or deadlocks, we are still safe, because the window manager will still be running. The end user can then kill the window. This will cause the quit dispatcher program to call Quit in the buffer and the adapter. When the buffer and the adapter terminate, the client and the application builder processes become isolated, and they will be garbage collected.

ApplicationBuilder:

using(Standard, StartWindowApplication, BBExternal, Adapter, Quit, QuitDispatcher)

process (Init: WindowApplicationIn)
declare
  Parms: WindowApplicationInterface; -- initialization call message
  Client: StandardOut; -- client process
  ParmString: Charstring; -- parameter to client process
  ClientPutLine: PutLineOut; -- PutLine from Client to Adapter
  ClientGetLine: GetLineOut; -- GetLine from Client to Buffer
  ClientGetPgm: GetProgramOut; -- GetProgram from Client to
  Adapter
begin
  receive Parms from Init;
  block
    declare
      AdapterQuit: QuitOut; -- adapter's quit port
      BufferQuit: QuitOut; -- buffer's quit port
    begin
      call (AdapterOut # (create of Parms.GetProgram("adapter")))[
        (Parms.WriteToWindow, Parms.WindowName, Parms.GetProgram,
         AdapterQuit, ClientPutLine, ClientGetPgm);
      call (BBout # (create of Parms.GetProgram("unboundedbuffer")))[
        0, Parms.InputToWindow, ClientGetLine, BufferQuit);
      call (QuitDispOut # (create of Parms.GetProgram("quitdispatcher")))[
        (AdapterQuit, BufferQuit, Parms.Quit);
      Client := create of Parms.GetProgram(Parms.ProgramName);
      ParmString := Parms.ParmString;
return Parms;
block begin
  call Client(ClientGetLine, ClientPutLine, ClientGetPgm, ParmString);
on (others)
  end block;
on (others)
  return Parms exception NotCreated;
end block;
end process

Note the on (others) exception handler. This is a shorthand notation, signifying "handle all exceptions which are not already explicitly handled". In this example, this handler will be entered if anything goes wrong trying to create the client. The handler will return a NotCreated exception. Once the client is created, we don't want exceptions returned by the client itself to cause a NotCreated exception. Therefore, we enclose the call to the client within an inner block with a null on (others) handler.

Since the code of the quit dispatcher, adapter, and buffer are known at compile-time, we need not call GetProgram to load these programs from the program library at run time. We do this using a program literal.

A literal is an expression with a compile-time known value. We have used string literals, like "Hello, World!", integer literals, like 0, and boolean literals, like 'true'. These literals are just like the literals of familiar languages, so we didn't bother explaining them.

A program literal is just program text delimited by process and end process. For example, suppose that instead of loading QuitDispatcher from the library, we used a program literal instead. Here is the relevant fragment of module ApplicationBuilder:

call (QuitDispOut # (create of process (Init: QuitDispIn))
declare
  Parms: QuitDispInterface;
  Quit1: QuitOut;
  Quit2: QuitOut;
  Quit: QuitIn;
  QuitCM: QuitInterface;
begin
  receive Parms from Init;
  new Quit; connect Parms.Quit to Quit;
  Quit1 := Parms.Quit1;
  Quit2 := Parms.Quit2;
  return Parms;
  receive QuitCM from Quit;
call Quit1();
call Quit2();
Don't confuse program literals with static bindings, which Hermes doesn't have. As we stated early in the tutorial, everyone with a static binding to `printf` has access to the same console. But if you and I each have a program literal denoting the `QuitDispatcher` program, we will still not have access to the same process. If we each instantiate the program, we'll have two separate processes running the same code. The only way we can share a process is if one of us instantiates the process, and passes a copy of the output port to the other.

Don't confuse program literals with nested procedures in block structured languages. In block structured languages, nested procedures can refer to variables in the outer procedure. This is not true here. A program literal is a completely separate program with a completely separate set of declarations. The program literal is treated as if it had been compiled as a separate module. It evaluates to an initialized, checked value of type `predefined!program`.

You may write a program literal out of line, separately compile it, and refer to it by name within another program. This is convenient if the same program text appears as a program literal in several programs. You don't want to maintain multiple copies of the same program literal, in case you wish to modify the program.

Here is the same code with the program literal out of line:

```
linking (QuitDispatcher)

... call (QuitDispOut # (create of process QuitDispatcher))
  (AdapterQuit, BufferQuit, Parms.Quit);
```

An out-of-line program name is made visible by means of a linking list, which appears right after the imports list at the beginning of the source module. In this example, we make the name `QuitDispatcher` visible by writing `linking(QuiDispatcher)`.

### 2.8.2 Adapter and Quit Dispatcher

Test yourself trying to write the adapter and quit dispatcher processes. Here is the code:

```
Adapter: using(Standard, Adapter, WMExternal, Quit)
process(Init: AdapterIn)
declare
  Parms: AdapterInterface; -- initialization call message
  Put: PutLineIn; -- PutLine service
  Get: GetProgramIn; -- GetProgram service
  Quit: QuitIn; -- Quit service
```
Window: WriteToWindowOut; -- access to window
GetProgramOut: GetProgramOut; -- access to GetProgram service
WindowName: Charstring; -- name of window
Running: boolean; -- true if process has not been shut down
GetProgramCM: GetProgramInterface;
PutCM: PutLineInterface;
QuitCM: QuitInterface;
begin
   -- initialization
   receive Parms from Init;
   new Put; connect Parms.ClientPutLine to Put;
   new Get; connect Parms.ClientGetProgram to Get;
   new Quit; connect Parms.Quit to Quit;
   Window := Parms.AdapterToWindow;
   GetProgramOut := Parms.GetProgram;
   WindowName := Parms.WindowName;
   Running := 'true';
   return Parms;
   -- service loop
   while (Running) repeat
      select
         event Put
            receive PutCM from Put;
            call Window(WindowName, PutCM.Line );
            return PutCM;
         event Get
            receive GetProgramCM from Get;
            call GetProgramOut(GetProgramCM.Name, GetProgramCM.TheProgram);
            return GetProgramCM;
         event Quit
            receive QuitCM from Quit;
            Running := 'false';
            return QuitCM;
         otherwise -- should not occur
            end select;
      end while;
   end process
QuitDispatcher: using(QuitDispatcher, Quit) process (Init: QuitDispIn)
   declare
      Parms: QuitDispInterface;
      Quit1: QuitOut;
      Quit2: QuitOut;
2. A Miniature System

```
quit: quitin;
quitcm: quitinterface;
begin
  receive parms from init;
  new quit; connect parms.quit to quit;
  quit1 := parms.quit1;
  quit2 := parms.quit2;
  return parms;
  receive quitcm from quit;
  call quit1();
  call quit2();
  return quitcm;
end process
```

2.9 Summary

In this section, we have built a complete miniature system. The system is simplified, but it illustrates many features of real systems. For example, it is a functional enhancement—windows—which can operate with client programs which were designed prior to the enhancement. It is transparent—that is, the client programs do not need to be changed to use windows. It achieves this transparency by exploiting Hermes’ ability to rebind any output port to an input port of matching interface. Furthermore, because the window system as a whole looks like a standard client, we can run the window system itself within one of its own windows to get nested windows.

The application builder illustrates access control. The client is passed an output port to an adapter which calls a particular window—therefore the client can be constrained to write only into the appropriate window. And the window manager can revoke an errant client’s access to the window by terminating the adapter.

The window manager illustrates how systems add or delete services dynamically. In this case, client applications and the ports to them are added and deleted.

This example illustrates how we divide the responsibility and knowledge among the processes of a system. The front end deals with input from the end user. It does string handling, and converts string commands to calls. The window manager handles the set of window applications—it primarily deals with the table of active windows. The adapter and buffer processes convert between different interfaces, and also allow a service to be cut off. The window system and application builder processes load and link together other processes.

If you understand how the window manager system is built, you understand the essential concepts and uses of Hermes. It should be a straightfor-
ward exercise to write more complex systems by applying the same principles. For example, you may want the window system to support a new style of clients, which have access to the Create service and can create new windows for their child processes to run in. You may want more realistic error handling. You may wish to try writing improvements on some of the examples illustrated here.

You also may wish to think of how you would have written these programs in your favorite programming language. Don’t forget that the programs must be robust—there must be no crashes due to exceeding a storage limit, for example. The programs must also be safe—clients must not be able to write on windows they don’t own. You have to close all the loopholes—like opening /dev—which might circumvent the security of the system. The system may be running for a long time—so you must make sure not to forget to free unused dynamically allocated storage. It would be instructive to estimate how much thinking you have to do to ensure a correct implementation in another language and compare it to what is involved in doing the job in Hermes.

**Summary:** In this section, we have mostly applied what we’ve learned in previous sections to a more practical problem. We have introduced a few new constructs: record type definitions, unordered tables, keys, selectors, every of, extract, position of (with a selector and with a selected element), the inspect statement, the DuplicateKey exception, the on (others) notation. From here on, you have nothing more to learn about Hermes except a more complete list of primitive operations, and a more refined description of what has been discussed informally. Good luck!
Type and Typestate Checking

C.A.R. Hoare, in his 1981 Turing Award Lecture, "The Emperor's Old Clothes", called upon language designers to enforce what he called security in programming languages, and criticized the recently adopted Ada design for not meeting this requirement. By security, Hoare meant "[t]he principle that every syntactically incorrect program should be rejected by the compiler and that every syntactically correct program should give a result or an error message that was predictable and comprehensible in terms of the source language program itself." In a secure language, a bug in module X should show up as bad output from module X, not as a crash, or a random implementation-dependent perturbation to a possibly innocent module Y.

Security in Hoare's sense is extremely important for a language like Hermes. Programs are long-lived systems, they include modules written by multiple users, they may dynamically load arbitrary untested programs, and they must keep on running the good programs even though some bad programs have been allowed to execute.

Let's adopt Ada's terminology and call executions which violate security erroneous executions. Erroneous executions are unchecked violations of language rules. (We're that the language has some concept of modularity in which it is illegal for one module to access the private data of another.) In practice, erroneous executions are the result of: (1) accessing undefined values, or (2) accessing data of the wrong type.

Conventional type checking avoids most of the problems of accessing data of the wrong type. The remaining security problems in languages like Ada are the result of aliasing and shared data. For example, two processes simultaneously write into a single variable, and the result is a possibly interleaved succession of bytes from both values, which may not be a well-formed value of the appropriate type. Or one process modifies a variant after another has already checked its discriminant. Since Hermes has no sharing and no aliasing, these pathological cases cannot arise.

The Hermes type system is straightforward. Every variable name has a type which is fixed throughout its scope, and which is directly derived from the declaration. Every operation has class rules, which limit which families of types the operands may have—for example, + may only be used with integer or real types. Every operation has type inference rules, which determine the type of one operand as a function of the type of another—for example if S is a Charstring, defined as ordered table of Char, then in insert C into S, C must be of type Char. The declared types of variables and the inferred types of expressions must (1) determine a unique type for
every expression and (2) be consistent with the type rules.

The problem of undefined values is solved in Hermes by typestate checking — a generalization of dataflow analysis. Hermes compilers use typestate checking to track where a variable is uninitialized (and therefore may not be read), where a variant is in the wrong case (and therefore may neither be read nor written), and where a program value is unchecked (and therefore may not be instantiated). A Hermes compiler will reject a program whose Pascal or Ada counterpart would compile successfully but produce an erroneous execution.

Here’s an example of typestate checking applied to a simple (and rather meaningless) program fragment:

```
block
declare
L/: Charstring;/
A/: Char;/
B/: Charstring;/
C/: Charstring;/
begin
 call Parms.GetLine(L);
 if size of L /= 0
 then
  A /:= /'NUL'/;
  B /:= /"Empty Line"/;
  call Parms.PutLine(B);
 else
  A /:= L[0]/;
  call Parms.GetLine(C);
  call Parms.PutLine(C);
 end if;
 insert A into L;
 call Parms.PutLine(L);
call Parms.GetLine(L);
on (GetLineInterface.EndStream)
call Parms.PutLine(L);
end block;
```

For now, let us look only at the simplest application of typestate — keeping track of which variables are initialized: A typestate is represented as a set of attributes describing properties of variables, such as whether a variable is initialized or uninitialized. If variable A is initialized, the attribute init(A) is present in the typestate; otherwise the attribute is absent. Typestate analysis computes a typestate for each program point. This means that at a given program point variable A must be known to be either initialized or uninitialized. We do not allow A to be initialized along some paths and not others.
Let us follow the analysis of typestates in the above example. To simplify the discussion, we will look only at typestate attributes relating to the initialized state of variables $A$, $B$, $C$, and $L$.

On entry to the block, none of the variables is initialized, so the typestate is empty. After the call to GetLine, the typestate is $\{\text{init}(L)\}$—only $L$ is initialized. At the end of the then clause, after the assignments to $A$ and $B$, the typestate is $\{\text{init}(L), \text{init}(A), \text{init}(B)\}$. At the end of the else clause, the typestate is $\{\text{init}(L), \text{init}(A), \text{init}(C)\}$.

At the end of the if statement, the two paths merge. The typestate at the merge point is computed as the intersection of the typestates. In this case, the typestate is $\{\text{init}(L), \text{init}(A)\}$—variables $L$ and $A$ will be initialized, $B$ and $C$ uninitialized.

Here is where typestate analysis goes beyond dataflow analysis. In dataflow analysis, you always know less information at a merge point than at either entry to the merge. So all you would know about $B$ and $C$ would be that they were possibly initialized, and possibly not. In Hermes, we do not wish there to be run-time tests of initializedness—a variable must either be known to be initialized or known to be uninitialized. The intersection rule implies that $B$ and $C$ will be uninitialized at the merge point. To make sure that $B$ and $C$ are in fact uninitialized at this point regardless of the execution path taken, the compiler inserts coercion operations. In this case, the compiler will insert a discard $B$ statement after the then clause and a discard $C$ statement after the else clause.

A typestate with a strict subset of the attributes of a second typestate is said to be lower. Coercion operations always convert from a higher typestate to a lower one.

A similar situation prevails at the exception handler on $\text{GetLineInterface.Endstream}$. There are three paths to that handler—one from each call to GetLine. The three typestates are $\{\}$ (first call), $\{\text{init}(L), \text{init}(A)\}$ (second call), and $\{\text{init}(L)\}$ (third call). The intersection is the typestate $\{\}$. The compiler automatically inserts operations to discard $L$ and $A$ into the path from the second call to GetLine to the handler, and it inserts an operation to discard $L$ into the path from the third call.

Typestate analysis not only tracks the typestate and performs coercions, it also checks for legality of programs. For instance, at the insert statement, it is checked that both $L$ and $A$ are initialized. In the statement call $\text{Parms.GetLine(A)}$, the $\text{GetLineIn}$ definition requires the argument to be uninitialized, yet $A$ is initialized. Here, the typestate is too high. A coercion operation discard $A$ is inserted so that $A$ will be uninitialized as expected.

---

1Mathematically, the set of typestates is a semilattice—that is, a mathematical structure where the elements (typestates in this case) are partially ordered, and where every pair of elements has a meet or greatest lower bound. In Hermes, one typestate is lower than the other if it is a subset of the other; the meet of two typestates is their intersection.
before the call. In the statement `call Parms.PutLine(L)` in the handler, `L` is required to be initialized, and it is not, so the typestate is too low. There is no coercion to make a typestate higher, so this statement must be rejected as erroneous. The compiler will issue an error message saying that attribute `init(L)` was required but not present.

What benefits do we get from typestate checking? First, we get security. Allowing a statement like `call Parms.PutLine(L)` to execute with an undefined `L` could produce unpredictable results—possibly even a program trap. (Imagine `L` implemented as a pointer to a string buffer.) This would be fatal, especially if our program is a multi-user system. And if we didn’t detect the error at compile-time, we’d have to check for it at run-time. Second, we get early detection of nonsense programs. Third, we get efficiency. When programs are written in unsafe languages, each user has to be put in a separate address space so that one user’s erroneous program can’t hurt another user. But we can run multiple Hermes processes in one address space. Because the implementation is secure, we get the protection of address spaces at the run-time cost of lightweight processes. And on machines lacking memory mapping, we get protection which we would not otherwise have. Finally, we get free garbage collection in the form of automatically generated discards of values. For instance, we saw that when the `EndStream` exception was raised, the appropriate variables were discarded before entering the exception handler.

What price do we pay? You have to declare the pre-call and post-call typestates of call parameters on interfaces. This can be viewed as either a burden or a benefit depending upon your point of view. If you’re picking up someone else’s modules, the extra documentation on the interface is useful. If you’re writing a procedure which no one but yourself is going to use and you can keep the interface “in your head”, you may consider writing extra declarations a burden. Hermes is designed primarily for large, long-lived systems, so it favors the first point of view over the second.
4

Additional Hermes Constructs

The examples in chapters 1.1 and 2 illustrate most of the novel Hermes constructs. However, there are some useful Hermes constructs which have not yet been discussed, so we will briefly discuss them here:

4.1 Expression Blocks

Hermes is a statement-oriented language. However, expressions are so convenient and so widely used that it would be foolish to bar them from Hermes for the sake of uniform syntax. An expression is an operation or series of operations which compute a single value which is immediately used as the operand of another statement or expression. All the operands of an expression except the result are unmodified by the expression.

Sometimes it is desirable for an expression to include embedded statements. We support this in Hermes with an expression block. We'll illustrate this by rewriting the Create service of the window manager.

```
receive CreateCM from Create;
if exists of CreatedWindows[CreateCM.WindowName]
then
   -- error action if window already exists
else
   block begin
      insert
         (evaluate W : Window from
            new W;
            W.WindowName := CreateCM.WindowName;
            call (WindowApplicationOut # (create of ApplicationBuilder))
               (CreateCM.ProgramName, CreateCM.ParmString,
               GetProgram, WriteToWindowCapability, W.WindowName,
               W.InputToWindow, W.Quit);
         end)
      into CreatedWindows;
      on (WindowApplicationInterface.NotCreated)
         -- error action if unable to create process
      end block;
   end if;
return CreateCM;
```
In this version of the program, the block consists of a single insert statement. The operand of insert is the record CurrentWindow which is built within the expression block. In this particular case, the expression block wasn’t really necessary. The original version was just as convenient. However, in some contexts, such as after where in a selector, you must use an expression block if you need to write a statement.

4.2 Send

Another common idiom is to send some data to another process and not wait for a reply. You can do this by defining an interface in which the data you wish to send is initialized on entry and uninitialized on exit. However, there is still the problem that the receiving process may choose to wait indefinitely before receiving your data. It may even run forever and never receive your data.

A possible Hermes solution is this: the sending process, S creates an intermediate process I. S calls I, which immediately receives the data, and returns to S. Then I calls the eventual receiver R. If R is delayed or deadlocked, it is I who is blocked and not S.

 Rather than requiring the programmer to explicitly code an intermediate process, Hermes provides an asynchronous communication mechanism. The statement send variable to port causes the value currently in variable to be removed and sent to the queue at the input port connected to port. Because of the send operation, input ports can be defined to hold any datatype, not just callmessages.

Here are two practical uses of send:

- **Communicating with untrusted processes**: Suppose I wish to load and execute a process and pass it some initialization information—possibly some ports. I could pass this information with a call statement, just as we did in earlier examples. However, there is some risk to doing this if I do not trust the process I am loading. Perhaps the process will never return my callmessage, and I will block forever. To be safe, I either must spawn a process to initialize and call the untrusted process, or I must deliver the initialization parameters using a send statement.

- **Forwarding calls**: Suppose I am writing a process whose job it is to receive calls and route them to an appropriate process which will actually service the call. If my process receives the callmessage and then calls the eventual destination, it must wait until the call is serviced. However, if instead my process transfers the callmessage using send, my process is free to route other calls. When the eventual destination issues a return statement, the callmessage will be returned to the original caller, and not to my process.
4.3 Variants

Sometimes the requirement that the type of every value be known at compile-time is too constraining. Hermes supports variants and polymorphs to allow this requirement to be weakened. However we do not give up the fundamental principles that machine-level implementation detail must be hidden, and that the language must be secure.

We illustrate variants with an example which will be familiar to anyone who has programmed in LISP. A LISP object is either an atom (which for purposes of this example, we'll assume to be just a character string printname), a pair, or nothing (nil). Here are the Hermes types which let us talk about LISP objects:

\[
\begin{align*}
\text{LispKind} & : \text{enumeration ('atom', 'pair', 'nil')} \\
\text{LispPair} & : \text{record(Car : LispObject, Cdr : LispObject)} \\
\text{LispObject} & : \text{variant of LispKind ('atom' -> PrintName : Charstring {full}, 'pair' -> Pair : LispPair {full}, 'nil' -> Nil : Empty {full})}
\end{align*}
\]

The enumeration and record types are just like similar types in other languages. Empty is a predefined type—an enumeration with no values—which is used conventionally to represent a null value. We'll explain the variant type.

Suppose \( L \) is a variable of type LispObject. Then \( L \) has three components: \( L.PrintName \) of type Charstring, \( L.Pair \) of type LispPair, and \( L.Nil \) of type Empty. In this respect, a variant is just like a record. But whereas an initialized record may have several components, only one component of a variant can exist at any one time. Which component it is depends upon the case of the variant. The case is an enumeration value—in this example, either 'atom', 'pair', or 'nil'. Each component is associated with one of the cases of the variant. So the value of an initialized variant is a case value plus exactly one of the variant components.

An initialized variant can be either hidden or revealed. Hidden means that we don't know the case at compile time. Revealed means we do. This is a typestate property. That means that the compiler will keep track of where in the program the variant is hidden, and where it is revealed. When it is revealed, the compiler tracks its case. So if \( L \) is a LispObject, the possible typestates of \( L \) are:

- uninitialized
- initialized and hidden
- initialized and revealed in case 'atom'
- initialized and revealed in case 'pair'
• initialized and revealed in case 'nil'

There are two ways to initialize an uninitialized variant. You can use an assignment statement to copy the value of another variable of the same type which is already initialized. Or you can build a new variant. When you build a new variant, you must decide which case to put the variant in. You may not directly assign to a component of an uninitialized variant. You must instead use the `unite` statement to move a value into one of the variant components. The `unite` statement simultaneously: (1) sets the case of the variant, (2) moves the value of its operand into the variant component, (3) tells the compiler that the variant is now initialized and revealed.

1. Here is an example of a statement which initializes L into case 'atom':

   ```lisp
   unite L.Printname from "Foo";
   ```

2. This statement puts L into case 'nil':

   ```lisp
   unite L.Nil from Emptyval;
   ```
   (Emptyval is a variable of type Empty. It doesn't need to be initialized, since in the definition, component Nil is expected to be uninitialized.)

3. This statement puts L into case 'pair':

   ```lisp
   new Pair;
   unite Pair.Car.Printname from "Foo";
   unite Pair.Cdr.Nil from Emptyval;
   unite L.Pair from Pair;
   ```

The typestate attribute `case` is present when the variant is revealed. For example, when variant L is revealed in case 'atom', the attribute `case(L, L.PrintName)` is present. This means that `L.PrintName` is the component which is known to exist.

When you write the definition of the variant type, you have to supply a typestate for each variant component. This is the typestate the variant component will have (1) just before it becomes hidden, and (2) just after it is revealed. It is also the minimum typestate the component can have any time it is revealed — that is, it may be made more initialized but not less.

The definition of LispObject illustrated above shows that `L.PrintName` must be fully initialized when L is in case 'atom', `L.Pair` must be fully initialized when L is in case 'pair', but `L.Nil` need not be initialized at all when L is in case 'nil'.

The typestate after each of the three `unite` statements above is respectively:

1. `{init(L), case(L, L.PrintName), init(L.PrintName)}`
2. \{\text{init}(L), \text{case}(L, \text{L.Nil})\}

3. \{\text{init}(L), \text{case}(L, \text{L.Pair}), \text{init}(\text{L.Pair.Car}),
\text{case}(\text{L.Pair.Car}, \text{L.Pair.Car.PrintName}), \text{init}(\text{L.Pair.Car.PrintName}),
\text{init}(\text{L.Pair.Cdr}), \text{case}(\text{L.Pair.Cdr, L.Pair.Cdr.Nil})\}

Notice that in case `pair`, \text{L.Pair.Car} and \text{L.Pair.Cdr} are themselves variants of type LispObject. The definition requires that when \text{L.Pair} is revealed, both \text{L.Pair.Car} and \text{L.Pair.Cdr} must be fully initialized. In the example, they are both initialized and revealed. This is legal, since fully initialized is the minimum typestate, and initialized and revealed is a higher typestate. It would be illegal to perform an operation to make \text{L.Pair.Car} uninitialized while \text{L.Pair} is revealed.

If the case attribute is not present, then the variant is hidden. You may hide a variant explicitly with the hide statement. But usually variants become hidden because of a coercion. For example, if you wrote a program which took three different paths, each executing one of the unite statements mentioned above, then at the merger of the paths the typestate would be \{\text{init}(L)\}.

When the variant is initialized but hidden, you may not access any of the components, because only one of them really exists and you don't know which one it is. For example, suppose the typestate is \{\text{init}(L)\} and you try to do the assignment \text{L.Printname} := "Foo". This will generate a typestate error message saying "adding attribute init(L.PrintName) would result in impossible typestate". The typestate \{\text{init}(L), \text{init}(L.PrintName)\} is impossible because you may not have any attribute of a variant component without the appropriate case attribute.

To reveal a variant, use the reveal statement. The operand of reveal is the component you hope to access. If the component does not exist because the variant is in the wrong case, you will get a CaseError exception. If you want to know the case of a variant so you know which component it is correct to reveal, issue the case of operation. This returns the enumeration value of the case. Here's a fragment of a procedure which swaps Car and Cdr of a LispObject to arbitrary depth:

```
receive Parms from init;
block begin
    reveal SwapCM.L.Pair;
    Temp := Swap(SwapCM.L.Pair.Car); -- it's a pair
    SwapCM.L.Pair.Car := Swap(SwapCM.L.Pair.Car);
    SwapCM.L.Pair.Cdr := Temp;
    on (CaseError) -- it's not a pair
    end block;
return Parms;
```

After the reveal, the component of the revealed variant has the typestate attributes which the definition said it was supposed to have. You may raise
the typestate (i.e. add attributes), but you may not lower any attributes. Furthermore, any attributes you added will be coerced away before the variant is hidden again. This guarantees that when the variant is revealed again, it will have a known typestate regardless of what you did. So in the above example, `Parms.L.Pair` will be fully initialized, since the definition of LispObject said that in case 'pair', component `Pair` is full. This makes the typestate

```
{init(Parms.L), case(Parms.L, Parms.L.Pair),
  init(Parms.L.Pair), init(Parms.L.Pair.Car), init(Parms.L.Pair.Cdr)}
```

If you wrote `discard Parms.L.Pair.Car`, this would be illegal. You would get a compiler error message saying that attempting to drop `init(Parms.L.Pair.Car)` would yield an impossible typestate. If you raise the typestate, for example by issuing `reveal Parms.L.Pair.Car.PrintName`, the typestate will be lowered back automatically as soon as you try to hide `Parms.L`.

4.4 Polymorphs

A variant is useful when you know ahead of time a fixed set of alternative types (or typestates) a value will have. Sometimes there is an open-ended set of alternatives. In such a case, you have to use a polymorph instead of a variant.

A polymorph is a variable which can hold any type of value. The value is always stored together with its type and typestate. Think of the value as being wrapped in an opaque wrapper labelled with the type and typestate. When the value is wrapped, it can be stored in a polymorph variable, inserted into a table of polymorphs, etc. But none of the operations of the original type are allowed. In order to issue these operations, the polymorph must be unwrapped and its value stored back into a variable of the appropriate type. At this time, there is a run-time check that the type and typestate you are expecting match the type and typestate appearing on the wrapper.

The only operations you may perform on a polymorph besides wrapping and unwrapping are those operations allowed on all types, such as inserting into a table. It is syntactically legal to copy a polymorph, but you will get the exception `Uncopyable` if the wrapped value is one of the uncopyable types. You may also inspect the wrapper to read the type and typestate of the wrapped value.

Here is a simple application of polymorphs—a resource repository into which the owner of a resource (e.g. a port into a server) may post a copy of the resource and a resource name, and authorized users may retrieve a copy of that resource by name. Since different resources are of different types, the interface to the repository cannot specify any particular type. Instead we require the owner to wrap the resource into a polymorph before
posting the resource. The requestor of a resource supplies a resource name and receives back a polymorph, which the requestor must then unwrap.

The owner of a resource may not wish to allow any arbitrary user to gain access. To provide access control, we require the owner to provide a port into an access control function. The access control function accepts a user name and the resource originally posted, and then either returns the resource being granted to the user, or returns a Denied exception.  

Here is the interface:

RepositoryExternal: using() definitions
Resource: polymorph;
PostInterface: callmessage(
    ResourceName: Charstring, -- name being given to resource
    PostedResource: Resource, -- resource being posted
    AccessFunction: AccessOut) -- access control function
    constant(ResourceName, AccessFunction)
    exit {full}
    exception DuplicateName {full};
PostIn: import of PostInterface {full};
PostOut: outport of PostIn;

AccessInterface: callmessage(
    UserName: Charstring, -- name of the requestor
    PostedResource: Resource, -- resource originally posted
    ReturnedResource: Resource) -- the resource granted
    constant(UserName, PostedResource)
    exit {full}
    exception AccessDenied {init(UserName), init(PostedResource)};
AccessIn: import of AccessInterface {init(UserName),init(PostedResource)};
AccessOut: outport of AccessIn;

RequestInterface: callmessage(
    ResourceName: Charstring, -- name of resource being requested
    Resource: Resource) -- the resource granted
    constant(ResourceName)
    exit {full}
    exception NotFound {init(ResourceName)}
    exception AccessDenied {init(ResourceName)};
RequestIn: import of RequestInterface {init(ResourceName)};
RequestOut: outport of RequestIn;

\[\text{The resource returned may or may not be the same as the resource posted. The access control function may decide to grant limited access to the resource through a filter, for example, in which case it will return a port into the filter rather than the port into the resource.}\]
Here's a program fragment which posts the GetLine service with trivial access control — anyone can get it.

Trivial := procedure of process (Init: AccessIn)
declare
  Parms: AccessInterface;
begin
  receive Parms from Init;
  Parms ReturnedResource := Parms PostedResource;
  return Parms;
end process;

wrap copy of GetLine as Resource;
call Post("GetLine", Resource, Trivial);

The wrap statement removes the value of its source operand before wrapping it. If you want to use GetLine after the wrap statement, be sure to wrap a copy.

Here is the code to obtain the GetLine port which someone else has posted:

block begin
  call Access("GetLine", Resource);
  unwrap GetLine from Resource {init};
  ...
  on(PolymorphMismatch)
  on(others)
end block;

The PolymorphMismatch exception will be raised by the unwrap statement if the type of the wrapped value of Resource doesn't match the type of variable GetLine, or if the typestate of the wrapped value is lower than {init}. These checks are necessary to preserve security. Without the checks, you could wrap a value supporting one set of operations and then later unwrap it and perform different operations.
Research Directions in Hermes

If you have followed the examples so far, you should be reasonably fluent in Hermes.

We believe that Hermes is an excellent technology for large system development. It combines expressive power, simplicity, portability, and safety, merging the best features of many language paradigms: Like object-oriented languages, Hermes supports information hiding and program reuse—but unlike object-oriented languages, there is support for concurrency and access control ([NS88, Str86]). Like LISP, Hermes allows running programs to build other programs, but unlike LISP, Hermes will check these programs for type errors before execution. Like Ada and Modula, Hermes supports modularity and interface specification, but unlike these languages, Hermes provides complete security using typestate checking. Like C/UNIX, Hermes supports systems programming, but uses a simpler and completely machine-independent set of primitives to do it. And Hermes provides the familiar sequential programming paradigm of the imperative languages.

As of 1990, the Hermes project at IBM Research is using Hermes to explore newer approaches to simplifying the development of complex systems:

- A more refined semilattice structure, which allows programmers to track more information about the value of variables, and which would allow additional coercions ([SY89, SY90]).
- Transparent recovery, which makes it unnecessary for programmers to treat process state as volatile and do their own checkpointing and restart ([SY85, SY84, YSB87, SBY88, SBY87, SYB87c, SYB88, FBSY87, SYB87d]).
- Optimistic transformations, which perform concurrency control ([Jef85]), process replication ([GJ87]), call-streaming ([BS90]), and parallelization transparently ([SY87]). Such optimizations make it unnecessary to code these complex protocols by hand. Straightforward but superficially inefficient programs, such as a server with thousands of clients coded as a single serial process, are transformed into efficient implementations on parallel and distributed architectures.
5. Research Directions in Hermes
Part II

Hermes Reference Manual
Introduction

The reference manual presents the rules of Hermes. Most of the rules are defined in machine-readable tables. We explain how to read the tables. We explain the other rules informally but rigorously. The manual contains examples of how to apply the rules, but these examples are not necessarily typical or preferred uses of Hermes.

We assume that you have already read and understood the Hermes tutorial. We also assume that you have experience with some other programming language. Therefore, we will not explain basic concepts such as stored programs, compilation, BNF, etc.

Here are the stages of writing and executing a Hermes program.

- You write the code using your favorite editor/program generator, producing a source file.
- The first stage of the compiler converts your source file into a Hermes program value.
- The Hermes compiler checks the Hermes program, producing a checked program value. The internal representation of a checked Hermes program will include its translation to machine code. Like all internal representations, this will be invisible to the programmer.
- You store the checked program into a program library.
- You retrieve the checked program, instantiate it as a process, and execute it.

You may also generate a Hermes program value directly within a Hermes program. This bypasses the first two steps mentioned above.

In this manual, we ignore the first step because we do not care how you produce the source file. The remaining steps are broken down as follows:

- **Lexical analysis:** The characters in the program are divided into tokens. Space and comment tokens are thrown away, and the other tokens are passed as input to syntax analysis. The lexical rules are written in a formal grammar. Lexical analysis is explained in chapter 7.

- **Syntax analysis:** The stream of tokens is parsed, revealing the structure of the program. The syntax rules are written in a formal grammar in Appendix A.
- **Conversion and resolution**: The parsed program is converted into a Hermes program value. This requires the compiler to resolve all names. Names are made up of identifiers. For each identifier there is a defining occurrence where the identifier is associated with its definition. Other appearances of the identifier are called applied occurrences. Resolution means associating each applied occurrence with a defining occurrence. The resolution rules are described informally in chapter 8.

- **Type checking and inference**: The type of each variable must satisfy the type rules for the operation where the variable appears as an operand. For variables without declarations (expression temporaries), the compiler applies type inference to generate a declaration. It is checked that exactly one type can be inferred and that this type satisfies the type rules. The type rules are given in an appendix. The type checking and inference algorithm is explained in chapter 9.

- **Typestate checking**: The typestate at each point in the program is computed. It is checked that the typestate at each operation is consistent with the typestate requirements of that operation. Coercions are generated where necessary. The typestate rules are given in an appendix. The typestate checking algorithm is explained in chapter 10.

- **Other static restrictions**: These are constraints on the value of a program other than those captured in the table-driven rules. Example: The number of arguments to `call` must match the number of components in the `callmessage` type. These restrictions are described informally next to the constructs they restrict.

- **Execution**: We describe the semantics of each operation rigorously but informally in chapter 11.
Lexical and Syntactic Rules

Lexical analysis separates the source program into tokens. Tokens are collections of characters that constitute the elementary syntactic units. The different kinds of token in Hermes are: spaces, comments, integer literals, real literals, named literals, string literals, punctuation, and words.

Spaces are used to separate tokens. Comments are used for human documentation, but to the computer they are equivalent to spaces. There are two commenting styles. A double dash “--” starts a comment that runs to the end of the line. Comments within a line begin with “/\*” and end with “\*/”. Comments delimited with “/\*” and “\*/” may not extend across multiple lines. Because of this rule, you need not look at a context bigger than a line to see what is a comment and what is not. To “comment out” a set of statement lines, precede each line with a double dash.

A literal token defines a compile-time known value.

- An integer literal token is a decimal representation of an unsigned integer. Some examples of integer literals are 0, 112358, and 000. Notice that \(-2\) is not a token. It is parsed as a unary operator applied to an integer literal.

- A real literal consists of an unsigned integer part, a decimal point, a fraction part, an “e” (upper or lower case), and an optionally signed integer exponent. The integer part must have at least one digit, but the fraction part can be empty. The symbol “e” means “times ten raised to the power of”. Some examples are 15., 0., 1.12358, 0.314159e+02 and 314.159e−02.

- A named literal is represented by a nonempty string of characters enclosed in two single quotes such that, inside the delimiting quotes, single quotes always occur in pairs. The value of a named literal is the string of characters inside the delimiting quotes where every pair of consecutive single quotes is regarded as one single quote. The case of the characters is significant in named literals. Some examples of named literals are ‘true’ and ‘Isn’t’. A named literal cannot extend past a single line.

- A string literal is represented by a (possibly empty) string of characters enclosed in two double quotes such that, inside the delimiting quotes, double quotes always occur in pairs. The value of a string literal is the string of characters inside the delimiting quotes where every pair of consecutive double quotes is regarded as one double
quote. The case of the characters is also significant in string literals. Some examples of string literals are "" (the empty string), "quoth e", and "A " symbol begins a quotation". A string literal cannot extend past a single line.

Punctuation consist of one or two characters. They are used for some operator names and for grouping and delimiting syntactic units. Some examples are parentheses – ( and ) – and the move operator: <-. The legal punctuation tokens are presented in Table A.2 in Appendix A.

A word is a string of characters that are alphanumeric characters or the underscore character. A word cannot begin with a numeric character. Words are used either identifiers (programmer-defined names) or keywords (words serving as syntactic markers). Some keywords cannot be used as identifiers. They are called reserved words. The case of the characters of a word token is ignored. Some examples of identifiers are AbCd_123, foo, and _x. The keywords and reserved words are listed in Table A.1 in Appendix A.

Hermes's syntax is given in Appendix A. It is written in BNF, using the yacc ([ASU88, AU77, KP84]) notation.
Resolution

Resolution means matching applied occurrences of names with their defining occurrences, and rejecting undefined applied occurrences or conflicting defining occurrences.

The following classes of names appear in Hermes: variable name, type name, attribute name, exception name, exit name, and process name. Names consist of one or more identifiers. Each identifier in an applied occurrence of a name must be resolved.

For each class of name, there are rules specifying:

- the name space in which the name is defined. A name space is an association between identifiers and their definitions. Within each name space there can be only one defining instance of a particular identifier. (Remember that case is ignored, so `x` and `X` cannot appear in the same name space.)

- the regions where each name space is visible — that is, where the identifiers can be referred to by applied occurrences.

- how to disambiguate a name if the name being resolved is in more than one visible name space. It is a resolution error either if there is no visible definition, or if there is more than one visible definition and no disambiguation is possible.

A name is said to be **builtin** if its meaning is given in the Hermes language definition and there is no defining occurrence. For example, the typestate attribute `init` and the exception name `NotFoundError` are builtin. A name is said to be **predefined** if it is defined in the `predefined` module, which is implicitly imported by all Hermes modules. For example, the type names `Charstring`, `integer`, and `program` are predefined. Programmers do not have to supply definitions for either builtin or predefined names.

### 8.1 Variable Names

A variable name denotes a container for a value. Variable names are used as operands.

```plaintext
variable-name ::= base-variable [ . component-name ] ...
```
8.1. Variable Names

base-variable ::= identifier


8.1.1 Base Variables

Base variables are defined within namespaces called declarations lists. The region of visibility of a declarations list is called its scope. Scopes appear only within executable code — either process modules or constraint definitions. Scopes can be nested. In an executable module, the process body is the outermost scope. The declarations list comprises the the initialization port declaration and the base variables declared following the declare keyword. The block statement defines an inner scope, in which the programmer can declare base variables. The selector, the inspect, the for statement, and the expression block also introduce scopes containing a single base variable declaration. Scopes also appear within constraint definitions. The outermost scope in this case is the declaration list of the constraint definition.

It is illegal for the same identifier to be defined within two overlapping scopes. (In other languages, this inner declaration would "hide" the outer declaration producing a "hole" in the outer variable's scope.) It is permissible for the same identifier to be declared twice within a program if the scopes do not overlap. Example:

block
  declare
    X: integer;
    T: table of integers;
  begin
    for X in T [] -- illegal redeclaration of X
      inspect
        call PutLine(IntToString(X));
      end for;
  end block;

for T in Z where(T > 0) -- legal redeclaration of T
  inspect
    call PutLine(IntToString(T));
  end for;

The resolution rule for base variables is trivial: A base variable is visible within its scope. Because base variables cannot have overlapping scopes, there can be a maximum of one visible defining occurrence of any base variable.
8.1.2 Component Names

Component names are defined in record, variant, and callmessage type definitions. Each record, variant, or callmessage type definition defines a name space for component names. This implies that a record type R cannot have two different components named X, but two different record types can each have a component named X.

Variable names containing components are resolved from left to right. First the base variable is resolved. Then the component of a variable is resolved within the component name space associated with the variable’s type definition. For example, in the name X.Y.Z, I first resolve the name X. Then I look up X’s type. I look up that type’s definition to find the name space within which I can resolve Y. I can then look up Y’s type, and repeat the process to obtain the resolution of Z. Note: a type may be known even when the type name itself is not visible — e.g. in the above case the type of X.Y might not be visible, because the definitions module did not appear in the using list.

8.2 Type Names

Type names are used in variable declarations, in type specifiers, and in definitions. Type names are written either simple identifiers (type identifiers), or they are type identifiers preceded by a definitions module name.

\[
\text{type-name} ::= \text{definition-name} \\
                 ::= \text{module-name}!\text{definition-name}
\]

\[
\text{definition-name} ::= \text{identifier}
\]

Type identifiers are defined within a definitions module—either a definitions module written by the user, or the predefined module. The set of type definitions within a definitions module forms a name space—that is, two type definitions within the same definitions module must have distinct names.

Within a definitions module, the following type definitions are visible: (1) those of the definitions module itself, (2) those in predefined, (3) those in definitions appearing on the imports list.

The imports list appears at the header of every module. It lists the names of definitions modules whose definitions will be made visible.

Within an executable module, there are no type definitions, so only type definitions of categories (2) and (3) are visible.

A type name not preceded by a module name is legal if and only if there is exactly one visible definition of that type.
If a module name is specified, the type name refers to the definition within the specified module. The module name must be either predefined, or a definitions module on the imports list.

8.3 Attribute Names

Attribute names are used in typestates. The syntax of attribute names is identical to the syntax of type names.

The names init and case are built-in typestate attributes. The name full is a built-in abbreviation for a set of init attributes. The name checked denotes a built-in constraint attribute. The name checkeddefinitions denotes a built-in attribute described in section 11.10. All other attribute names are names of constraints defined in definitions modules. The set of constraint definitions of each definitions module forms a name space. That name space is different from the space of type definitions, so it is legal to define a type T and a constraint T in the same module.

Except for the special treatment of the five built-in names, the resolution rules for attribute names are identical to the resolution rules for type names.

8.4 Exception Names

Exception names are used in exception handlers, and on the return statement. Exception names are either built-in, or user-defined. The two have different syntax.

Built-in exception names denote the exceptions associated with Hermes primitive operations. A built-in exception name is written as a simple identifier. The identifier must be one of the names in the predefined type predefined!builtin_exception.

User-defined exception names denote the exceptions returned from call statements. They are written as:

```
exception-name ::= type-name . user-exception-name
```

On the return statement, the type name is understood to be the same as the type of the callmessage operand, so only the exception identifier is written. There is a name space of exception identifiers for each callmessage definition. That name space contains the user-defined exceptions, plus the automatically defined exception discarded. A user-defined exception name is resolved simply by resolving the type name. The type name must be a callmessage type. This determines a single name space which is used to resolve the exception name.
8.5 Exit Names

The exit statement allows the programmer to jump to a handler clause without raising a builtin exception or issuing a call. Exit names are defined on the handler and used on the exit statement.

The defining occurrence of an exit name is a handler beginning on exit. The name space is the identifier list appearing after the word exit. The name is visible throughout the main clause of the block containing the handler. Unlike base variables, there can be several overlapping regions of visibility for the same exit name.

The applied occurrence of an exit name is on an exit statement. When resolving an exit name, if there are several visible definitions of the same exit name, the innermost visible definition is used.

8.6 Process Names

A process name is used only in the program literal construct:

```
program_literal : process_identifier
```

The identifier must be present in the linking list of the source module. The identifier denotes the process module of that name.
Type Checking and Inference

After a module has been tokenized, parsed, and resolved, it is stored as a Hermes program value. This is a much simpler form as far as the computer is concerned. All the “noise words”, like of, from and into, which make the program easier for people to read but less syntactically regular, are gone. Statements with embedded expressions are expanded out to sequences of statements. Names, like $X$, whose resolution may be context-sensitive, have been removed. In their place are unique identifiers.

The Hermes program value is: (1) a collection of type definitions and constraint definitions, organized into definitions modules; (2) a collection of declarations, organized into “scopes”; (3) a collection of clauses. Each clause is a series of statements, and each statement is an operator, a list of operands (variable names), and a “qualifier”, which is everything else. By this time, all base variable names except expression temporaries have been declared.

The job of the next phase is: (1) type inference: to supply declarations for the expression temporaries so that every base variable will have a known type; (2) type checking: to check that the operations are legal, given the types of the operands.

Both type inference and type checking are driven from a single table of rules. There are two families of rules: inference rules, and class rules. The inference rules are used for both inference and checking. The class rules are an extra set of checks which apply after type inference is complete.

The rules appear in the appendix. The syntax of an inference rule is:

```
inference_rule : operand ← rule_name ( )
```

The first kind of rule unconditionally assigns a type to the operand on the left. The second kind of rule is fired when the type of the operand on the right becomes known. The rule provides a function which determines the type of the operand on the left.

The type rules in the table entry for this operation look like this:

```
inert_at( table, element, position ) absent
  table ∈ orderedtable
  position ← predefinedinteger()
  element ← elementtypeof( table)
```

For example, let’s look at the operation insert_at. The header line states that the operation insert_at has three operands, which will be referred to in the rules as table, element, and position. There is no qualifier. The
header line is followed by three type rules. The first rule is a class rule, which states that the type of table must be an ordered table type. The second rule says that the operand position—this is the operand after the word at—is unconditionally inferred or checked to be of type predefined!integer. The third rule says that if the type of the table is known, then the type of the element will also be known by applying the function elementtypeof.

The type checking algorithm begins by computing all known types—the types of those operands which are variable names, and of those expression results which were explicitly marked with a type specifier. For each operation, the inference rules with empty right-hand sides, or whose right-hand side operand is known are fired. Each time an inference rule is fired, it infers the type of the operand on its left side. If that operand currently has known type, the inferred type is checked against the known type. A mismatch is an error. If the operand currently has unknown type, the inferred type is assigned by generating an inferred declaration for the operand. Now that the type of this operand is known, this may cause other rules to fire.

Type checking continues until no additional firings are possible. By now, all variables should have been assigned types. If any variables still have unknown types, the program is in error and the compiler will flag each occurrence of such variables.

There are some variables which, for technical reasons, are not considered operands, but are instead grouped with the qualifier. Type inference and type checking are also performed for these variables, but those checks are not table-driven. They are noted in the appendix as comments. For example, the test variables used in if, while, select, and the predicate in a selector (the expression following the keyword where) are inferred as predefined!boolean. Because of this inference, it is legal to write expressions like if (a > b), which otherwise would be ambiguous since the operation > can return a result of any boolean type. This inference can lead to detection of type errors elsewhere, e.g. if (3) ... will produce a message that the integer-literal operation cannot yield a result of type predefined!boolean.

After all types have been inferred, the class rules are invoked. A class rule is a constraint on the possible types a given operand can have. The class rule is applied regardless of whether the operand’s type was inferred or originally known. For example, the table operand of the insert at operation must be an ordered table. There is no class rule for the element operand, since it may be of any type. Since an inference rule already guarantees that the position operand is of type predefined!integer, a class rule would be redundant.

Note: Type checking and inferencing is performed after resolution. Type correctness is never used as a criterion for disambiguating a name.

Here are four examples illustrating type inference and checking. Assume that type Charstring is a table of Char, and that Queue is a table of Charstring.
block
declare
  X: Charstring;
  Q: Queue;
  I: Integer;
begin
  new Q;
  insert "abc"|"cd" into Q; -- inference from top down
  X := "uvw";
  I := size of (X | "abc"); -- inference from bottom up
  insert X | "abc" into Q; -- inference in both directions
  I := size of "abc"; -- no inference; illegal
end block;

The relevant operations are: (1) string literal—the class rule says the
type must be a string, but there is no inference rule; (2) size of—there is
an inference rule for the result type (predefined integer), but none for the
source type; (3) insert—the type of the element can be inferred from the
type of the table; (4) concatenation—there are two sources and a result, all
of the same type, so if any one type is known the other two can be inferred.

In the first statement, the type of "abc"|"cd" is inferred to be Charstring
because the type of Q is known to be Queue. The type of "abc" and "cd"
are inferred to be Charstring because the rules for concatenation can now
be fired.

In the second statement, the type of X | "abc" cannot be inferred from
size of, because even though the result type is known to be integer, there
is no inference rule to infer the source type. But since the type of X is known
to be Charstring, the rule for concatenation determines that the types of
"abc" and of the result, X | "abc" must be Charstring, too.

In the third statement, the inference can work in both directions. Because
the type of Q is known, the rule for insert can fire to determine the type of
X | "abc". Because the type of X is known, the rule for concatenate can fire
to determine the type of X | "abc". Whichever rule fires first will set the
type; whichever rule fires second will check the type. If the types inferred
in the two directions had been different, an error would have resulted.

In the fourth statement, no type inference is possible. The string literal
operation doesn't infer the type—it only has a class rule. (Hermes has no
Ada-like rule which tries to search the space of all visible types to see
whether only one legal type is visible.) The size of operation likewise has
no inference rule for the source operand. Since there is no type assignment
to "abc", the statement is illegal—it needs to be rewritten I := size of
Charstring # "abc".

The prefix charstring # is called a type specifier. The effect of a type
specifier is to explicitly declare a type for an expression operand. When a
type specifier is used, the temporary variable is treated as having known
type rather than unknown type. This known type still must be consistent
with the inference rules and class rules. The example above shows that sometimes a type specifier is required. Type specifiers may also be used for documentation purposes even where they are not required.
10

Typestate Checking

Typestate checking assigns a typestate to each program point. A typestate is a mathematical object which can be interpreted in two ways: (1) as an assertion that certain properties of program variables will be true whenever that statement is executed; (2) as a point in a semilattice \(^1\). Because typestates are assertions, they can be used to check at compile-time that certain language rules expressed in terms of those assertions are satisfied. Because typestates form a semilattice, certain well-known algorithms from dataflow analysis can be used.

In this section, we will discuss the particular typestate semilattice used by Hermes. We will describe the typestate checking algorithm we use. We will tell you how to read the table which defines the particular typestate rules for each operation.

10.1 Syntax of Typestates

Typestates are explicitly written in the following situations: (1) in interface definitions (input ports, callmessages, and constraints), (2) in table and variant definitions to indicate what typestate an element or variant component must have, (3) in the unwrap statement, to indicate what typestate a polymorph must have.

A typestate is written as a set of attributes separated by commas and enclosed in curly braces. An attribute is an attribute name followed by a list of variable names called attribute arguments.

\[
\text{typestate ::= \{ attribute , attribute \ldots \}}
\]

\[
\text{attribute ::= attribute-name attribute-arguments}
\]

Examples of attributes: \text{init(X,A)}, \text{case(V,V,A)}, \text{greater_than(X,Y)}.

The following attribute names are built-in: \text{init}, \text{case}, \text{checked} and \text{checked definitions}. Constraint attributes, like \text{greater_than}, are user-defined.

The arguments of builtin attributes are checked as follows: Attribute \text{init} must have a single argument, and \text{case} must have exactly two argu-

\(^1\)A semilattice is a set with a partial order relationship \(\leq\) such that any pair of elements \(t_1\) and \(t_2\) have a unique \textit{greatest lower bound} or \textit{meet} \(t\) such that \(t \leq t_1, t \leq t_2\) and for any \(t'\), \((t' \leq t_1 \land t' \leq t_2) \Rightarrow (t' \leq t)\).
ments. The first argument to case must be a variant variable, and the second must be a component of the same variant variable. Attribute checked must have a single argument of type predefined ! program. Attribute checkeddefinitions must have a single argument of type predefined ! definitions module.

All other attribute names are constraint attributes. When you mention a constraint attribute, there will be a constraint definition which specifies how many attribute arguments there must be and what the type of each must be.

For example, if there is a constraint definition

greater_than: constraint (a: integer, b: integer) IS
  {init(a), init(b)} a > b;

then the attribute greater_than must always appear with exactly two arguments of type integer.

The name full is syntactically an attribute name, but it is treated as an abbreviation. If you write full(X), and X is a record or callmessage, then it’s the same as if you wrote init(X) and full(X,A) for each component X.A of X. For other types, full means the same as init. From now on, we’ll assume full has been expanded out wherever it appears. Because of this use as an abbreviation, you may not define an attribute named full.

Here are the meanings of the typestate attributes:

- init(X) means the variable X is initialized; otherwise it is uninitialized.
- case(V, V.X) means that variant V is revealed, and component V.X exists; the absence of case(V, ...) means that variant V is hidden.
- checked(P) means that program P has been checked and is free of syntax errors, including resolution errors, type errors, and typestate errors.
- checkeddefinition(D) means that definitions module D has been checked and assigned a unique name. (See the discussion of the checkdefinitions statement.)
- attributename(arguments) means that the named constraint predicate is known to be true of its arguments.

10.2 Formal Typestate

Sometimes a typestate must be written without the base variables. For example, here is the definition of a record, and a table of these records:

R: record(A: integer, B: Charstring);
The definition says that the typestate of an arbitrary variable \( V \) of type \( R \) before being inserted into a table of type \( T \) must be \( \{ \text{init}(V), \text{init}(V.A), \text{init}(V.B) \} \). When we write the definition, we omit the arbitrary variable \( V \). The resulting typestate is called a \textit{formal typestate}. A formal typestate looks like a typestate, except that we omit the base variable from each argument of each attribute, turning it from a variable name into a \textit{formal variable name}. If the variable name is a base variable with no components, it is replaced by \(*\). The argument list \((*)\) can optionally be omitted.

Formal variable names are also used in key lists, where they denote components of an arbitrary element of a table.

A formal variable name or formal typestate is always written in a context in which the type of the arbitrary variable is known.

To substitute a variable name into a formal typestate means to prefix each formal variable name with the given variable name.

For example, if variable name \( X.Y \) has type \( R \), then you can substitute \( X.Y \) into the formal typestate in the definition above, and obtain the typestate \( \{\text{init}(X.Y), \text{init}(X.Y.A), \text{init}(X.Y.B)\} \). You can substitute \( X.Y \) into the key and obtain an actual key of \( X.Y.A \).

### 10.3 Valid Typestates

Because of the meaning of the Hermes data types and the meaning of the attributes, not every combination of attributes is valid.

Here are the \textit{attribute compatibility} rules which define certain combinations as invalid:

- If \( \text{init}(X) \) is not present, no other attribute with argument \( X \) may be present. This follows from the fact that only \( \text{init} \) variables can have values.

- If \( X \) has components, and \( \text{init}(X) \) is not present, then no other attribute with any component as argument may be present. This follows from the fact that components of structured values only exist when the structured value itself exists.

- If \( \text{case}(V, V.A) \) is present, then no other attribute \( \text{case}(V, V.B) \) is present. This follows from the fact that only one component of a variant exists at a time.

- If \( \text{case}(V, V.A) \) is absent, then no other attribute with \( V.A \) as component may be present. This follows from the fact that you can access components of a variant only when they are revealed.

- If \( \text{case}(V, V.A) \) is present, then \( V.A \)'s \textit{case typestate} attributes are present. These are the attributes obtained by substituting \( V.A \) into
the formal typestate defined for component A in the type definition for variant V. This follows from the fact that we require these attributes to be present when we hide the variant, and we require that the typestate of a revealed variant must be higher than the typestate of an initialized but hidden variant.

- If \( \text{init}(CM) \) is present and \( CM \) is a callmessage, then the attributes in the \textit{minimum typestate} of \( CM \) must be present. These attributes are obtained by substituting \( CM \) into the formal typestate defined or implied as the minimum typestate. (See discussion of callmessages.) This is because the programmer is allowed to state in the interface that certain call parameters must remain initialized and cannot be discarded by the process receiving the call.

Otherwise legal operations whose effect would be to produce an invalid typestate are illegal—e.g., an assignment to record component \( R.A \) when \( R \) is itself uninitialized.

### 10.4 Ordering of Typestates

The semilattice structure is simple. A typestate is lower than a second typestate when the set of attributes of the first is a subset of the set of attributes of the second. The \textit{meet} of two typestates is simply the intersection of the attribute sets.

Example: \{\text{init}(A)\} is lower than \{\text{init}(A), \text{even}(A)\}. The meet of \{\text{init}(V), \text{case}(V, V.A), \text{init}(V.A)\} and \{\text{init}(V), \text{case}(V, V.B), \text{init}(V.B)\} is \{\text{init}(V)\}.

### 10.5 Coercions

\textit{Coercions} are typestate-lowering operations which are inserted automatically by the compiler. There are three coercions:

- \textit{discard} \( X \) removes the attribute \text{init}(X), as well as any other attributes of \( X \) or \( X \)'s components.

- \textit{hide} \( V \) removes the attribute \text{case}(V, V.X), along with any other attributes of \( V.X \).

- \textit{drop constraint}(P1, ..., Pn) removes the attribute \text{constraint}(P1, ..., Pn).
10.6 Constants

Although we use the term "variable" to denote a cell which can hold a value and appear as an operand, sometimes "variables" are "constant"! That is, sometimes it is only legal to access the value of a variable but not to change it.

A variable is constant at a particular program point if:

- It is a component of an initialized callmessage, and the callmessage type definition lists that component as a constant.
- It is a base variable, and an enclosing block statement lists that base variable as constant.
- It is a base variable which was initialized to a constant copy. (See selector, inspect, for statements.)
- The program point is inside an expression block and the base variable was declared outside that expression block.
- It is a component of a constant.

10.7 How to Apply the Typestate Rules

Here are the steps in checking a single statement and propagating a new typestate to its successor statements:

1. Determine the precondition for the operation. There is a table giving a set of precondition rules for each operation. These rules determine:
   (1) a set of required attributes, (2) a set of forbidden attributes, (3) the operands modified by the operation, (4) possible additional exceptions.

2. If any required attribute is missing, this is an error.

3. Introduce coercions which drop attributes which are present but are forbidden. If we cannot add coercions without also dropping required attributes, then this is an error. Otherwise, we have succeeded in generating a typestate which is consistent with the preconditions.

4. Generate an error if any variables modified by the operation are constants.

5. Determine the set of outcomes of the operation. There is a "normal" outcome for all operations (except exit), and there can be several "exception" outcomes. The table entry for each operation defines which exception outcomes the operation has. The table may list additional exceptions which the operation has under certain circumstances.
— for example, it may say that the operation has a `DuplicateKey` exception outcome provided its operand is a keyed table.

6. Determine the postconditions for each possible outcome of the operation. For each outcome, use the rules given in the table to determine which attributes to add to or drop from the typestate.

7. Check that after adding and dropping the appropriate attributes the resulting typestate is valid.

8. When each typestate is computed, propagate that typestate to the destination. If that destination already has a typestate, compute the meet (as defined above) of its previous typestate and the typestate you have just computed for this path to that destination. Insert coercion operations to lower the typestate to that meet typestate.

Let’s apply these steps to an example. Suppose I have these type definitions:

\[
\text{NamedProgram: record(A: Charstring, B: Program);}
\]

\[
\text{ProgramRepository: table of R \{full\} keys(A);}
\]

Now suppose I is of type `integer`, P is of type `NamedProgram` and R is of type `ProgramRepository`. Suppose the current typestate is: \{init(I), init(R), init(P), init(P.A), init(P.B), checked(P.B)\}. Suppose the statement being checked is:

\[
\text{insert P into R;}
\]

Look at the table of operation rules in Appendix B. After the class rules, there are three rules relevant to typestate checking. These are: the precondition rules, the postcondition rules, and the exception list. For `insert`, the precondition rules are:

\[(\text{init(table)}, \text{lowestelementstate(element, table)}, \text{var(table)}, \text{var(element)}, \text{duplicatekey?(table)})\]

The first two members of the list are typestate precondition functions. Rule `init(table)` says that the table operand must be initialized. Rule `lowestelementstate(element, table)` says that the element operand must be in the lowest typestate consistent with the element typestate of the table. (Element typestate is defined below in section 11.6.) The element typestate is \{full\}, which is an abbreviation of \{init(*), init(A), init(B)\}. Substituting P into this formal typestate yields \{init(P), init(P.A), init(P.B)\}. Any attributes of P or its components other than the above attributes are forbidden. Section B.4 explains how to apply each precondition function to determine the required and forbidden attributes.

As a result of applying the precondition rules, it is determined that these attributes are required: \{init(R), init(P), init(P.A), init(P.B)\}. This attribute is forbidden: checked(P.B). Since all the required attributes
are present in the typestate, there is no "missing attribute" error. However, there is an "extra" attribute—\texttt{checked(P.B)}. The coercion \texttt{drop \checked(P.B)} is inserted to drop the \texttt{checked(P.B)} attribute. This causes the compiler to forget that the program value \texttt{P.B} has been checked, because it is being inserted into a table containing program values which are initialized but not checked. When the value is removed from the table, it will be assumed to be unchecked. The typestate is now \{\texttt{init(I)}, \texttt{init(R)}, \texttt{init(P)}, \texttt{init(P.A)}, \texttt{init(P.B)}\}.

The rules \texttt{var(\textbf{table})} and \texttt{var(\textbf{element})} tells us which operands are modified by the operation. From this rule, we infer that \texttt{P} and \texttt{R} must not be constant. If they are, the compiler flags an error.

The "rule" \texttt{duplicatekey?(\textbf{table})} is not a precondition rule. For technical reasons, it is included with the precondition rules and evaluated when preconditions are evaluated. The effect of the rule is to check whether the table \texttt{R} was defined as keyed, and if so, to include \texttt{DuplicateKey} in the set of exceptions which could be raised.

The possible outcomes of \texttt{insert} are: normal, \texttt{Depletion}, and \texttt{DuplicateKey}. The \texttt{Depletion} exception was determined from the exception list, because every \texttt{insert} operation can raise this exception. The \texttt{DuplicateKey} was derived by evaluating the predicate \texttt{duplicatekey?(\textbf{table})}, because only insertions into keyed tables can raise this exception.

The postcondition for normal exit is determined by the postcondition rules. For \texttt{insert}, these rules are:

\begin{verbatim}
(makeuninit(element), killconstraints(table))
\end{verbatim}

Except for the call statement, all exception outcomes produce an unchanged value and an unchanged typestate—as if the operation had not executed. So the typestate associated with the two exception outcomes is also \{\texttt{init(I)}, \texttt{init(R)}, \texttt{init(P)}, \texttt{init(P.A)}, \texttt{init(P.B)}\}. For this reason, exception postconditions are not shown in the table.

The rule \texttt{makeuninit(element)} makes \texttt{P} uninitialized by dropping all attributes of \texttt{P} and its components—namely \{\texttt{init(P)}, \texttt{init(P.A)}, \texttt{init(P.B)}\}. The rule \texttt{killconstraints(table)} drops any constraint attributes of the table—there were none. The postcondition functions are explained in section B.5.

So the typestate associated with normal completion of \texttt{insert} is: \{\texttt{init(I)}, \texttt{init(R)}\}.

10.8 The Checking Algorithm

The initial typestate is one in which the initialization port is initialized and no other typestate attributes are present.

At each statement, the rules in the table are used to determine the required and forbidden attributes prior to the operation, and the added and
dropped attributes after normal completion operation.

Three types of statements are handled specially:

- The call statement is different because the normal and exception postconditions come from the interface definition.

- The compound statements have more complex postcondition rules because they are equivalent to miniature flow graphs. For instance, an if statement behaves as if it were composed of a more primitive boolean test statement and the embedded clauses. In the reference manual we describe the typestate postcondition resulting from this flow graph.

- Similarly, every statement containing a selector is analyzed as if it were two statements: a selector followed by a primitive statement. The rules in the table define the precondition and postcondition rules for the primitive statement; these rules are applied after the rules for selectors.

Exceptions, exit statements, and the branches from the end of clauses within block, if and select statements to the end of the statement are forward branches. When a forward branch is encountered, the meet is taken between the previous typestate associated with the destination and the typestate propagated from the statement performing the forward branch.

Each loop construct (while, for) performs a single backward branch from the end of the loop to the beginning. If the typestate at the end of the loop is strictly greater than or equal to the typestate at the beginning, nothing is done except to introduce any necessary coercion. If the typestate at the end of the loop is lower than or incomparable to the typestate at the beginning, the meet is taken between the two typestates. This lowers the typestate previously associated with the top of the loop. The statements of the loop are then re-analyzed.

Analysis continues until all program points have been assigned a typestate and no loops need to be re-analyzed. In practice, no loop will need to be analyzed more than twice.

10.9 Typestate Errors

The possible typestate error messages are:

- **Dead code**: A statement cannot be reached. Remember that typestate checking ignores the values of variables, so that even a statement beginning if 'false' then ... does not generate a dead code exception. This error arises when you write a statement after an exit statement or after a compound statement all of whose alternatives
terminate with exit statements. It also arises if you code a handler which can never be branched to.

- **Attribute not present**: A required typestate attribute was not present. For example, you coded an insert operation but the table was not initialized.

- **Cannot coerce**: The operation requires an attribute to be dropped, but the coercion which drops the attribute would also drop another attribute which is required. Example:

```plaintext
call Service(Parms.A,Parms.B);
```

Suppose the interface to Service requires the first argument to be initialized and the second to be uninitialized. Suppose Parms is a callmessage whose minimum typestate requires both Parms.A and Parms.B to be initialized. Then the call statement requires Parms.B to be made uninitialized, but this cannot be done without discarding Parms, which would then make Parms.A uninitialized.

- **Cannot add**: The postcondition rule mandates adding this attribute, but to do so would produce an invalid typestate. For example, an assignment to Parms.A without first initializing Parms would produce this message.

- **Cannot drop**: The postcondition rule mandates dropping this attribute, but to do so would produce an invalid typestate. For example, dropping a callmessage below its minimum typestate.

The following, while not technically typestate errors, are static context-dependent errors which are detected during typestate checking:

- **Illegal constant**: You have coded a statement which modifies one of its operands, but the operand is constant.

- **Illegal position**: You have coded the position of operation in a context where it is not legal. This operator is legal only when the table element operand is a constant copy.
11

Hermes Operations

Throughout the remainder of the reference manual, syntactic rules will be excerpted from Appendix A as specific language constructs are introduced. Low-level syntactic elements that appear in rule bodies generally will not be included in the excerpts. The grammar is presented in its entirety, including these low-level elements, in Appendix A.

11.1 Ubiquitous Operations

Ubiquitous operations are those which apply to all or nearly all types.

Move

\[
\text{move} \quad \text{simple-statement} \\
\quad \quad \quad ::= \text{result-variable} \leftarrow \text{source-expression}
\]

The move statement removes a value from one variable \(^1\) (the source) and places it in another (the destination). If the destination variable had a value, that value is discarded. The source variable becomes uninitialized.

Example:

\[\text{R.A} \leftarrow X;\]

The source and destination variables must have the same type. Hermes uses "name-equivalence" for deciding whether two types match; therefore, types with identical domains or identical structures don't match.

Values of any type or typestate except totally uninitialized can be moved. However, neither source nor destination can be constant. Any constraints (e.g., \text{greater than}(X, Y)) of the source become constraints of the destination (e.g., \text{greater than}(\text{R.A}, Y)). Any previous constraints on the target \text{R.A} or on a containing variable \text{R} are dropped.

Throughout the manual, if we say "a value is moved" this means that the value no longer exists in its old place. This is distinct from saying "a value is copied", which means that the value remains in its old place, but a copy also exists in the new place.

\(^1\)Here and elsewhere, the term "variable" denotes any operand. Some "variables" are unnamed temporary variables holding the results of expressions. Some "variables" are constants.
11.1. Ubiquitous Operations

Copy

\[
\begin{align*}
\text{simple-statement} & \quad ::= \text{result-variable} := \text{source-expression} \\
\text{secondary} & \quad ::= \text{copy of secondary}
\end{align*}
\]

The copy of statement copies a value from one variable to another. The source variable retains its original value and typestate attributes. As with move, the destination variable loses its old value and attributes, but acquires the value and attributes of the source variable. The source variable may have any typestate except completely uninitialized.

There are two ways of writing a copy statement: as an expression like copy of X, or in an assignment statement like Y := X. The latter is equivalent to Y := copy of X.

As with the move statement, source and destination must have identical type.

Input ports and callmessages are uncopyable, as are aggregates containing uncopyable data, or polymorphs containing wrapped uncopyable values. If it is known from the type that a value is uncopyable, the copy operation will be rejected at compile-time. If it cannot be statically determined whether the variable is copyable (because the value includes a polymorph or variant), then the copy operation will be legal, but will raise the Uncopyable exception at execution time if the value is actually uncopyable.

Other operations besides copy of have the effect of copying values or parts of values. For example, every of produces a copy of a subset of a table. As with the copy operation, if the value's type is known statically to be uncopyable, then the statement is a compile-time error. If the type is not known statically to either be copyable or uncopyable, the Uncopyable exception will be raised at execution time if the value is in fact uncopyable.

Copies, or transitive copies of copies are always equal. Depending on the type, other values besides copies may be equal.

Equal, Not-equal

\[
\begin{align*}
\text{relation} & \quad ::= \text{concat} = \text{concat} \\
& \quad ::= \text{concat} <> \text{concat}
\end{align*}
\]

Values of any type, and of any typestate except totally uninitialized, can be tested for equality with the operators = and <>. For each type family, equality will be defined.

Operands being compared for equality must have the same type.
Discard

\[
\text{simple-statement} \\
::= \text{discard variable-name}
\]

The `discard` statement removes the value from the variable. Discarding an initialized callmessage returns the callmessage; the caller will receive the `callmessageertype.Discarded` exception. The callmessage components will first be lowered to their minimum typestate—this concept is defined later, in the section on callmessages. Discarding a value containing callmessages—for example an input port holding enqueued callmessages—discards the callmessages. Discarding any other type of value just throws the value away.

### 11.2 The Depletion Exception

Most operations can raise this exception, so it is described here rather than repeatedly under each operation.

An ideal Hermes machine has unlimited resources. Real machines have limited resources. The Depletion exception is raised whenever due to a resource limitation or an implementation restriction, the correct result cannot be computed.

Examples include: running out of address space, exceeding disk capacity limits, integer overflow. Implementations are encouraged to avoid giving Depletion exceptions rather than to require programmers to develop their own circumventions. For example, integers which don't fit in a hardware register could be represented by dynamic-precision integers. But we don't require all implementations to be this sophisticated, so we reserve the Depletion exception to inform the user that the implementation is inadequate to run this particular execution.

Coercion operations—e.g. `discard`—cannot fail with any exception. Hermes implementations must guarantee that storage depletion cannot occur while discarding or hiding a value.
11.3 Control Flow Operations

Block

\[
\text{compound-statement} ::= \text{block}
\quad \text{constant-section}
\quad \text{declaration-section}
\quad \text{begin}
\quad \text{handler}
\quad \text{end block}
\]

\[
\text{constant-section} ::= \text{constant} ( [ \text{base-variable} [ , \text{base-variable} ] ... ] )
\]

\[
\text{declaration-section} ::= \text{declare} [ \text{declaration} ; ] ...
\]

\[
\text{handler} ::= \text{on} ( [ \text{exception-name} [ , \text{exception-name} ] ... ] )
\quad \text{on exit} ( [ \text{exit-name} [ , \text{exit-name} ] ... ] )
\quad ::= \text{on exit} ( [ \text{exit-name} [ , \text{exit-name} ] ... ] )
\quad \text{statement} ; ...
\]

A block statement introduces a new scope. Within this scope, you can declare new variables, you can specify that the values of existing variables must remain constant, and you can provide handlers for exceptions or exits within the main body of the block.

The statement series after the word \text{begin} is the \text{main clause} of the block. The other clauses are the \text{handler} clauses. Each handler clause is preceded by the names of the exits or exceptions it handles. The abbreviation \text{others} stands for all exceptions which are not explicitly handled. \text{Restriction:} Two clauses of the same block cannot handle the same exit or exception.

An exception or exit raised within the \text{main clause} of a block causes control to transfer to the clause headed by that exception or exit name. If a statement is within the main clauses of two or more nested blocks containing handlers for the same exception or exit, control will transfer to the handler of the innermost block. Remember that exceptions raised within a \text{handler} clause cannot cause control to transfer to another handler clause of the same block.

If all statements of a clause complete without raising an exception or issuing an \text{exit} statement, then the clause is said to exit normally. If any clause of a block exits normally, then the block itself exits normally.

The body of a \text{process} statement is actually a block, although the words \text{block} and \text{end block} do not appear. A second block with an empty \text{on(others)} handler is implied around this block. This is necessary so that
there will always exist a target for exceptions raised within handlers of the process statement.

The typestate on normal termination of a block statement is the meet of the typestates on normal exit from all the clauses which can exit normally. If none of the clauses can exit normally, the block statement can not exit normally. The only way a clause cannot exit normally is if it ends with an exit statement or if it ends with a compound statement which itself cannot exit normally.

If

\[
\text{compound-statement} ::= \text{if test-expression then} \ [ \text{statement} ; \ldots \ ] \ [ \text{else} \ [ \text{statement} ; \ldots \ ]] \end{if}
\]

The if statement contains an expression—the test—and two clauses—the then and else clauses.

It behaves like the if statement of conventional languages: The test expression is evaluated to an initialized boolean value. If the value is 'true', the then clause is then executed; if the value is 'false', the else clause is executed. An omitted else clause is treated as an empty else clause.

The type and typestate rules are as follows: The test expression result is inferred to have type predefined!boolean. The test expression result must have typestate init. The typestate on entry to either clause is the same as the typestate after execution of the test expression. The typestate on normal exit of the if statement is the meet of the typestates for each clause which can exit normally; if no clause can exit normally, then the if statement can not exit normally.

While

\[
\text{compound-statement} ::= \text{while test-expression repeat} \ [ \text{statement} ; \ldots \ ] \end{while}
\]

The while statement contains an expression—the test—and a clause—the repeat clause.

It behaves like the while statement of conventional languages: The test expression is repeatedly evaluated to an initialized boolean value. If the value is 'true', then the repeat clause is executed, and then the while statement is re-executed. If it is 'false', the while statement terminates.

The type and typestate rules are as follows: The test expression result is inferred to have type predefined!boolean. The test expression result is checked to have typestate init. The typestate on entry to the test expres-
Control Flow Operations

The typestate on entry to the repeat clause is the typestate after execution of the test expression. The typestate on termination of the while clause is the same as the typestate after execution of the test expression.

Select

```
compound-statement ::= select [ select-expression ]
                             [ select-clause ] ...
                             otherwise-clause
                   end select

select-clause ::= boolean-guard [ statement ] ...
                ::= event-guard [ statement ] ...
                ::= event-guard and boolean-guard [ statement ] ...

event-guard ::= event import-variable

boolean-guard ::= where ( test-expression )

otherwise-clause ::= otherwise [ statement ] ...

select-expression ::= expression
```

The select statement consists of an optional expression, a set of select clauses, and an otherwise clause. A select clause consists of a guard and a clause. A guard can be

- a boolean guard,
- a conjunction of a boolean guard and an event guard naming an input port, or
- an event guard alone, which is treated as if a boolean guard of "true" had been coded.

Example:

```
select
  where (x > 0)
  y := x + y;
  event import_i
```
receive parms_1 from import_1;
event import_2 and where (x <= 0)
receive parms_1 from import_2;
otherwise
end select;

If there is an optional select expression, then it is evaluated and the value is stored in a temporary variable. Each boolean guard expression is replaced by a comparison between the expression and the temporary variable. Example:

```plaintext
select employee.salary
where (low_salary)
    employee.raise := 7;
where (average_salary)
    employee.raise := 5;
where (high_salary)
    employee.raise := 2;
otherwise
    employee.raise := 0;
end select;
```

The `select` statement begins execution by evaluating each boolean guard (including the implicit 'true' guards). The clauses whose boolean guards evaluated to true are said to be *enabled*; the clauses whose boolean guards evaluated to false are said to be *disabled*.

- If all clauses are disabled, the *otherwise* clause is executed and then the `select` statement terminates.
- If one or more clauses are enabled, and all enabled clauses have event guards, and the input ports have no connections, then the Disconnected exception is raised.
- If there is an enabled clause with either no event guard, or with an event guard whose input port is nonempty, then that clause is executed. If there are more than one such clause, a non-deterministic choice is made. The choice need not be *fair*. That is, if the `select` statement is executed infinitely often and if it is always possible to choose a particular alternative there is no guarantee that this choice will eventually be taken so long as other choices are possible.
- If all enabled clauses have event guards, and at least one of the input ports has a connection, but all input ports remain empty indefinitely, then the `select` statement will wait indefinitely.

In the first example, the second clause has an implicit true boolean guard, so it is always enabled. The first clause will be enabled if x > 0 and the third clause will be enabled if x <= 0. Since some clauses will always be
enabled, the otherwise clause will never be executed. If \( x > 0 \), then if no message ever arrives at \( \text{import} \), the first clause must eventually be executed. But if a message ever arrives at \( \text{import} \) then either the first or second clause may be executed. If \( x \leq 0 \), then if a message arrives only at \( \text{import} \), the second clause will execute; if a message arrives only at \( \text{import} \), the third clause will execute; if messages arrive at both ports, one or the other clause will execute; if a message arrives at neither port, the statement will block.

In the second example, there are no event guards, and the statement behaves like the choice statements of other languages. If \( \text{low\_salary} = \text{average\_salary} \), then a non-deterministic choice is made between the first two clauses.

Here are the type and typestate checking rules: Boolean guard expressions are inferred to be of type \text{predefined!boolean}. Event guards are checked to be of type \text{family!input\_port}.

All boolean guards and all event guards must have typestate attribute \text{init}. The typestate on entry to any clause is the typestate after evaluating all guards. The typestate on normal termination of the statement is the meet of the typestates on completion of all clauses which can exit normally; if none of the clauses can exit normally, then neither can the \text{select} statement.

Expression Block

\[ \text{secondary} ::= \text{evaluate declaration from [ statement ; ] ... end} \]

An expression block, or evaluate operator, is a way to embed statements within an expression. It contains a declaration of a result variable, and a clause containing statements to compute the result variable.

The expression block introduces a new scope containing the result variable. Since expressions are not supposed to have side effects, all variables declared outside the expression block are constant within the block.

The typestate after executing an expression block is the typestate on normal exit from the block. It is a "dead code" error if the clause does not have a normal exit. The block must initialize the result variable.

```lisp
for \text{LispObject} in \text{LispObjects} where
  (evaluate \text{StartsWithZ: boolean from}
    block begin
      reveal \text{LispObject.Pair};
      reveal \text{LispObject.Pair.Car.PrintName};
      \text{StartsWithZ := LispObject.Pair.Car.PrintName = Z;}
    on (CaseError)
      \text{StartsWithZ := 'false';}
  end block;
```
This program fragment looks up the Lisp variable which is a pair whose first member is an atom with printname matching Z. The test must be written as an expression inside \texttt{where}(). To perform the test, we must write a \texttt{reveal} statement, so we need an expression block in order to enclose this statement within an expression.

\textbf{Exit}

\begin{verbatim}
simple-statement ::= exit exit-name
\end{verbatim}

The \texttt{exit} statement terminates execution of a block as if an exception had been raised. In effect, an exit is a locally defined exception condition. Example:

\begin{verbatim}
block begin
  if X>200000
    then
      exit TooBig;
    end if;
  . . .
n on exit(TooBig)
    . . .
  end block;
\end{verbatim}

The statement includes an identifier which is an exit name. Control branches to a handler for that exit name. The handler chosen must be in a block whose main clause contains the \texttt{exit} statement. If there is more than one such handler, the innermost one is chosen. (This rule is repeated in the discussion of the \texttt{block} statement.)

\section{11.4 Scalar Operations}

Hermes has five scalar type families: nominal, enumeration, boolean, integer, and real. The scalar types families are similar to scalar types in other procedural languages. In this section we first describe how to declare types in these families, and then discuss the scalar operations.

\textbf{Scalar Type Definitions}

A \textit{nominal} domain is a set of values which have no relationship other than equality. They are used to generate distinct \textit{names}—hence the term \textit{nominal}. You can generate a new, distinct nominal value with the unique op-
Scalar Operations

You can copy nominals and test them for equality. You cannot perform ordered comparisons or conversions to integers. Two nominal values are equal if they are copies of the same generated value, otherwise they are unequal. A nominal type definition creates a new nominal domain.

\[ \text{type-construction} \]

\[ ::= \text{nominal} \]

Example:

TransactionId: nominal;

An enumeration domain consists of a finite number of values. An enumeration type definition defines a new enumeration domain, and associates a named literal with each value, e.g. 'blue'. The named literals of a single enumeration type definition must be distinct. This named literal can be used in expressions to generate the value it names. Enumeration values of the same type are equal if they are equal to the same named literal. Of course, the value 'blue' in one enumeration domain has nothing to do with the value 'blue' in a different enumeration domain, since only values of the same type can be compared.

\[ \text{type-construction} \]

\[ ::= [ \text{ordered} ] \text{enumeration} ( \]

\[ [ \text{named-literal} [ , \text{named-literal} ] ... ] \]

\[ ) \]

\[ ::= \text{variant of} \text{enumeration-type} ( \]

\[ [ \text{case-declaration} [ , \text{case-declaration} ] ... ] \]

\[ ) \]

Enumerations are either ordered or unordered, depending on whether the keyword ordered appears in the definition. Unordered enumerations support comparisons only for equality and inequality, just like nominals. Ordered enumerations also support the ordered comparisons <, >, <=, and >=. They also can be converted to integers. The first enumeration literal has the converted integer value 0, the next 1, and so on.

The following are some examples.

option: enumeration ( 'present', 'absent' );

color: enumeration ( 'red', 'blue', 'yellow', 'green' );

quality: ordered enumeration ( 'poor', 'fair', 'good', 'excellent' );

There is a predefined ordered enumeration type called Predefined!Character that contains all the characters of the ASCII character set in sorted order.

A boolean domain consists of two values, one representing "true" and the other "false". A boolean type definition associates the true and false values with distinct named literals.

\[ \text{type-construction} \]

\[ ::= \text{boolean} ( \text{boolean-association} ) \]
Boolean types support the same operations as unordered enumeration types. Additionally, the operations "and", "or", and "not" are supported. There is a predefined boolean type called `predefined!boolean` whose named literals are 'true' and 'false'. You can define your own boolean types with different literals. Example:

```plaintext
bit: boolean (true: '0', false: '1');
```

An integer domain consists of the mathematical integers. Each integer type definition defines a new integer domain.

```plaintext
type-construction
::= integer
::= real of accuracy integer-literal / integer-literal
```

Example:

```plaintext
apples: integer;
oranges: integer;
```

As usual, 5 in the `apples` integer domain can't be compared to 5 in the `oranges` domain. But you can convert between two integer domains using the `convert` operation.

In some languages "integers" are defined modulo the size of the largest number that can fit in a word of computer memory. In contrast, Hermes uses the usual mathematical definition of integer. The result of an integer operation is always either the mathematically correct result, or else no result and a Depletion exception.

A real domain consists of a set of safe numbers. Safe numbers are a discrete subset of real numbers spaced sufficiently close together to meet a user-defined accuracy requirement. For every real number, there exists a safe number such that the relative error (the distance between the safe number and the real number divided by the real number) is less than or equal to the accuracy requirement.

The accuracy is expressed as a fraction, e.g. `1/1000000`. Example:

```plaintext
velocity: real of accuracy 1/1000000;
```

Two other properties of approximate arithmetic often assumed by programmers are supported: (1) Zero is a safe number, and (2) integers are "preferred" as safe numbers, i.e. any pair of safe non-integers with an integer between them must have a safe integer between them.

The usual real arithmetic operations are supported; the expression syntax is identical to that shown for integers above. Approximate real arithmetic is performed as follows. The result is computed and a safe number nearest the exact result is used as the result. If there is more than one nearest safe number to a number, a nondeterministic selection is made.
11.4. Scalar Operations

Integer, Real, and Named Literals

Literals are expressions which compute a result defined at compile-time. They are treated as nullary operations for the purpose of type-checking. Examples: 23, 'blue', "Hello, World!".

You cannot tell the type of a literal just by looking at the literal—only whether it is an integer, real, string, or named literal. For example, the literal 2 is not necessarily of type predefined!integer. You must either write a type specifier or put the literal in a context so that the type checker can infer the type, e.g. X <- 2. Usually, this will always be the case. But remember that if you write an expression like 'fair' < 'good', you will get a type error. The compiler will not scan through the set of all visible types to see if there is a unique one in which this expression makes sense. You must use a type specifier.

Add, Subtract, Multiply, Divide, Unary-minus

```
term ::= factor
    ::= factor + factor
    ::= factor - factor

factor ::= factor * secondary
         ::= factor / secondary
```

Examples: A + B, -X, A * B. Addition, subtraction, multiplication, division, and negation (unary minus) are defined for integer and real types. On integers, truncating integer division is performed. Division by zero raises a DivideByZero exception.

Rem, Mod

```
factor ::= factor mod secondary
         ::= factor rem secondary
```

Examples: X mod 2, Y rem Z. The remainder and modulus operators are defined for integers. The remainder operation a rem b produces the remainder of a/b. The modulus operation a mod b produces a result that has the sign of b, an absolute value less than the absolute value of b, and that satisfies

\[ a = bn + (a \mod b) \]

for some integer n. Division, rem, and mod raise exception DivideByZero if the second operand evaluates to 0.
Less, Less-equal, Greater, Greater-equal

\[
relation ::= concat < concat \\
::= concat > concat \\
::= concat <= concat \\
::= concat >= concat
\]

Examples: \( X = 2, Y <> 3, A <= B \). The result is of boolean type (not necessarily predefined boolean).

And, Or, Not

\[
disjunction ::= conjunction \\
::= disjunction or conjunction
\]

\[
conjunction ::= relation \\
::= conjunction and relation
\]

\[
secondary ::= not secondary
\]

Example: \( \neg (A \land B) = \neg A \lor \neg B \) These operations are defined for all boolean types. They have the usual Boolean algebra interpretation. You cannot mix operands of different boolean types. But you can easily convert between boolean types by comparing a result to “true”:

```
block
  declare
    X: boolean;
    Y: bit;
    Z: bit;
  begin
    ..
    X := Y and Z; -- illegal because types don’t match
    X := (Y and Z) = ‘1’; -- legal
  end block;
```

Unique

\[
typed-primary ::= unique
\]

Example: \( Y <- unique \). This operation generates a unique value of a nominal type. The value is guaranteed to compare unequal to any previously defined value of that type.
For-enumerate

\[\text{compound-statement} \quad ::= \text{for enumeration-variable : enumeration-type repeat} \]
\[\text{[ statement ; ] ...} \]
\[\text{end for}\]

The for enumerate statement iterates over the elements of an unordered enumeration type. This statement consists of a declaration and a clause, where the declaration specifies a variable belonging to an unordered enumeration type. For every value in the domain of the enumeration type, the variable is assigned that value and the clause is executed once. Inside the clause, the variable is required to be constant. The order in which the values are chosen is nondeterministic. The following is an example.

\begin{verbatim}
for thiscolor: color repeat
  send color to outport;
end for
\end{verbatim}

Convert

\[\text{secondary} \quad ::= \text{convert of secondary}\]

The convert of operator has only one operand. It converts values from one ordered type to another. An example of its use is:

\begin{verbatim}
x <- convert of y;
\end{verbatim}

The three ordered domains are related as follows: \text{ordered enumeration} \subset \text{integer} \subset \text{real}

When converting between an ordered enumeration type and an integer type, the first named literal in the former corresponds to 0, the second named literal corresponds to 1, and so on. In converting between two real types, the result is a safe number in the target domain nearest to the source value. When converting from integer or ordered enumeration into real, the result is a safe number nearest to the integer corresponding to the value in the source domain. In converting from real into integer (ordered enumeration), the result is an integer (a value corresponding to the integer) nearest to the value in the source domain.

If the target domain is an ordered enumeration domain, exception \text{RangeError} is raised when the result lies outside the domain.
11.5 Record Operations

Record Type Family

type-construction
 ::= record (  
       [ component-declaration [ , component-declaration ]  
          ... ]  
    )

A record is a tuple of values called components. When a record is initialized, the component variables exist. The component variables may themselves be either initialized or uninitialized. When a record is uninitialized, the component variables do not exist, are considered uninitialized, and cannot be made initialized.

Two records are equal for comparison purposes when the same components are initialized and all initialized components are equal.

A record type definition specifies the name and type of each component, e.g.

```
clause: record (  
    id: clauseid,  
    statements: statements  
  );
```

New

The new operation creates an initialized record whose components are uninitialized. The previous value of the record, if any, is discarded.

```
simple-statement
 ::= new variable-name
```

Note: the two ways to initialize a record variable which is currently uninitialized—issue the new operation, and then initialize the components, or assign or move a pre-existing record value.

11.6 Table Operations

Table Type Family

type-construction
 ::= [ ordered ] table of element-type element-typestate  
    [ keys [ key ] ... ]

key  ::= ( [ formal-variable [ , formal-variable ] ... ] )

A table is a collection of values of the same type and typestate. The values are called the elements of the table, the type the element type, and the
typestate the element typestate. Tables can be used to represent bags, sets, strings, arrays, lists, and relations.

A table type definition contains an element type and a formal typestate representing an element typestate. It can contain the keyword ordered. A table type definition can also contain a set of keys. Each key is written as a list of formal variables. Restrictions: (1) The variables of a key must be initialized, (2) the element typestate must not be fully uninitialized. These restrictions guarantee that table elements can be compared, and that keys can be compared.

If a table has keys, any two elements of the table will have different values of all keys. Two elements differ in a key if they differ in any variable of that key.

The value of an ordered table is an ordered (possibly empty) set of elements. The first element of the table is said to have position 0, the next 1, etc.

Two ordered tables are equal if they contain equal elements in the same order.

Two unordered tables are equal if their elements can be put in one-to-one correspondence and the corresponding elements are equal. The following are some examples of table type definitions.

```
intbag: table of integer {init};
string: ordered table of char {init};
string_set: table of string {init} keys (*);
person: record (  
    name: string,
    address: string,
    id: integer
);

person_table: table of person {full} keys (name, address) (id);
```

Type intbag is a bag—it can hold any number of integers, including multiple occurrences of the same integer. Type charstring is like a string in conventional languages: it is a totally ordered bag. Type string_set has the additional specification that no two strings may be equal—thus it behaves like a set. Type person_table has the property that no two elements can have the same strings for both name and address, and no two elements can have the same value of id. This type acts like a two-way mapping between id numbers and name-address pairs.

New

```
simple-statement  
    ::= new variable-name
```
The operation new creates an initialized but empty table. The previous value of the table, if any, is discarded.

String Literal

A string literal is a self-defining character string value. The type of a string literal is an ordered table of enumeration or boolean values. Each character in the string literal must correspond to a single-character named literal of the enumeration or boolean type.

So if "0011010" is used as a string literal, its resolved type must be a string of a type which includes values named '0' and '1'—e.g., a type defined as ordered table of Bit.

Concatenate

\[
\text{concat} ::= \text{term} \\
                ::= \text{concat} \mid \text{term}
\]

This operation concatenates two ordered tables to produce a result containing a copy of the elements of the left operand followed by a copy of the elements of the right operand. The elements must be copyable.

Selector

\[
\text{selector} ::= \text{base-variable in table-variable} \\
                 \quad \text{where ( selector-expression )} \\
                 ::= \text{base-variable in table-variable} \\
                 \quad [ [ \text{expression} \mid \text{expression} ] \ldots ] \\
                 ::= \text{table-variable} [ [ \text{expression} \mid \text{expression} ] \ldots ]
\]

The selector is a syntactic construct used in several operations. There are three syntactic forms permitted by the syntax. The first form is called a long selector, the others mapping selectors. The mapping selectors are simply abbreviations for long selectors.

The variable name (table-variable) is the name of a table. The base-variable in a selector is called an element declaration. It defines a new variable name called the element variable, whose type is the element type of that table. This variable name is visible within the where expression. The expression is called the selector expression. It evaluates to a result which is inferred to be of type predefined!boolean.

At execution time, the selector is evaluated once for each element in the table. The element variable is set equal to a constant copy of the element. The elements for which the selector expression are true are said to be selected.

A mapping selector is a shorthand representing some common uses of selectors. If the mapping list is empty, then all elements are selected—it is
a shorthand for \texttt{where('true')}. If the mapping list has a single expression, and the table is ordered, then expression is assumed to be a position and the selected element is the element at that position. Thus \texttt{t[n]} is shorthand for \\
\texttt{e in t where(position of e = n)}. If the above rule does not apply, and the mapping list has \( k \) expressions, and one key has \( k \) variables, the \( k \) expressions are assumed to be comparison values for those keys. Thus if \( t \) has type \texttt{person\_table}, then \texttt{t[x, y]} is shorthand for \texttt{e in t where(e.name = x and e.address = y)}. If none of these rules apply, or if more than one key has the correct number of variables, the use of the mapping selector is illegal.

Selectors appear as part of several operations. The use of the selected elements depends upon the operation. The purpose of some operations is to select a single element; the purpose of others is to select an entire set. When the purpose is to select a single element, and no element is selected, exception \texttt{NotFound} is raised. If more than one element is selected, and the table is ordered, the earliest element is chosen. If the table is unordered, an arbitrary choice is made.

A selector by itself in an expression is the operation called \texttt{the\_element}. The result is a copy of the chosen selected element, or else \texttt{NotFound} is raised. The result has the element type and element type state. The element type must be copyable.

\textbf{Every}

\begin{verbatim}
secondary ::= every of selector
\end{verbatim}

The \texttt{every} operation returns a table containing a copy of all the selected elements. If the original table was ordered, the copied elements are in the same order. The result of \texttt{every} will be an empty table if no elements are selected. The element type must be copyable.

\textbf{Exists, Forall}

\begin{verbatim}
secondary ::= exists of selector
::= forall of selector
\end{verbatim}

The \texttt{exists} operation returns a true boolean value if at least one element is selected; else it returns false. The \texttt{forall} operation returns a true boolean value if all elements are selected; else it returns false.

\textbf{Insert, Insert-at}

\begin{verbatim}
simple-statement ::= insert element-expression into table-variable [ at position-expression ]
\end{verbatim}
The insert statement adds a new element to a table by moving in a value from a source variable. The source variable must be of the element type. The value must have the element type state. If it has a lower type state, the operation is illegal. If it has a higher type state, the type state is lowered. If the table is ordered, the element is inserted at the end of the table. If the table is keyed, the DuplicateKey exception is raised if the insertion would have violated the requirement of unique keys.

The insert-at statement is legal only for ordered tables. It behaves exactly like insert, except it has an additional operand of type predefined!integer, which specifies where to insert the element. If this position is negative, or greater than the number of elements in the table, the exception RangeError is raised. Otherwise, the element is inserted at the designated position. The positions of earlier elements in the table remain the same. The positions of later elements (if any) are increased by one. The exception DuplicateKey may be raised exactly as for insert, except that a RangeError exception takes precedence.

After either insert or insert-at, the element variable is uninitialized, since its value is moved and not copied.

Remove

\[
\text{simple-statement} \quad ::= \text{remove element-variable from selector}
\]

The remove statement has two operands: a destination element, which appears after the word remove, and a selector. The chosen selected element is removed from the table and moved into the destination variable. If there is no selected element, the exception NotFound is raised. The previous value, if any, of the destination variable is discarded.

Examples:

\[
\begin{align*}
\text{remove Char from String[0];} \\
\text{remove Min from I in IntBag where(forall of J in Intbag where(J >= I));}
\end{align*}
\]

The first statement removes the first character from the string, if there is one. If there is none, an exception is raised. The second statement removes the smallest integer from Intbag—the integer I such that all integers in Intbag are at least as large as I. It will raise an exception if the table is empty.

Extract

\[
\text{simple-statement} \quad ::= \text{extract table-variable from selector}
\]

The extract statement has two operands: a destination table, which appears after the word extract, and a selector. All chosen selected ele-
ments are removed from the table named in the selector, and moved into the destination table. If no elements are selected, the destination table is initialized to an empty table. The previous value, if any, of the destination is discarded.

Merge, Merge-At

\[
\text{simple-statement} \\
::= \text{merge table-expression into table-variable} \\
\text{at position-expression}
\]

The `merge` statement moves the contents of the source table into the destination table. The two tables are of the same type—meaning that the elements will be of the same type and typestate. On normal completion, the source variable will be uninitialized. If the table is ordered, the elements (if any) from the source table will be inserted at the end of the destination table, and their order will be preserved. If the table is keyed, the `DuplicateKey` exception is raised if the merge would have violated the requirement of unique keys.

The `merge-at` statement is legal only for ordered tables. It behaves exactly like `merge`, except it has an additional operand of type `predefined!integer` which specifies where to insert the source table elements. If this position is negative, or greater than the number of elements in the table, the exception `RangeError` is raised. Otherwise the elements are inserted at the designated position. The positions of earlier elements in the table remain the same. The positions of later elements (if any) are increased by the number of elements merged. The exception `DuplicateKey` can be raised exactly as for `merge`, except that a `RangeError` exception takes precedence.

Inspect-table

\[
\text{compound-statement} \\
::= \text{inspect selector begin} \\
\text{statement ;} \\
\text{end inspect}
\]

This statement makes a constant copy of the chosen selected element of a table. This copy is stored into a new variable — the `inspect variable` having the same name as the element variable of the selector. The inspect variable is automatically declared to be of the element type of the table, and to have a scope including the clause within the body of the `inspect` statement. If there is no selected element, the `NotFound` exception is raised.

Whereas it is illegal to make ordinary copies of certain values (callmessages, input ports, or values containing them), it is always legal to make a constant copy of any value. This is because the operations which would be "dangerous" (`receive`, `return`) when performed on ordinary copies are
not allowed on constant copies. A variable holding a constant copy is automatically made uninitialized when control leaves the scope of the variable.

The following is an example of the inspect statement.

```
inspect char in string
  where (char = 'X' and position of char > 2)
begin
  insert copy of char into string2;
end inspect;
```

Note the double use of the variable `char`—(1) within the where expression as the element variable of a selector, and (2) within the insert statement as the variable holding the constant copy. Note the differences between the inspect statement and the similar statement using the operation the-element.

```
c := char in string where(char = 'X' and position of char > 2);
insert c into string2;
```

The inspect statement introduces a declaration of `char`, whereas `c` must have been previously declared. The inspect statement works for all types, whereas the-element works only with copyable types. The inspect statement produces a constant copy, whereas the-element produces an ordinary copy. It is illegal to apply insert directly to `char` because it is constant, but it is legal to apply insert directly to `c`. The position-of-element operation can be applied to constant copies but not to ordinary copies.

The typestate on entry to the main clause of the inspect statement is the same as the typestate prior to the inspect statement except that the inspect variable is in the element typestate. The typestate on exit from the inspect statement is the same as the typestate on exit from the main clause, except that the inspect variable becomes uninitialized. If the main clause cannot exit normally, then the inspect statement cannot exit normally.

```
For-Inspect

compound-statement ::= for selector inspect
  [ statement ; ] ...
end for
```

The for-inspect statement iterates over the elements of a table or a selected subset of the table. The table must be initialized. The main clause (the clause following inspect) is executed once for each selected element. A variable with the same name as the element variable of the selector holds a constant copy of the selected element. The value of the table on entry to the for-inspect statement is used to determine the selected values; any changes to the table made from within the iterated clause will not affect which values are selected. If the table is ordered, the elements will be
selected in that order.

The typestate on entry to the main clause of a for-inspect statement is the meet of (1) the typestate in which the inspect variable is in the element typestate and other typestate attributes are the same as they were before the for-inspect statement, (2) the typestate at the end of the main clause, assuming it can exit normally. The typestate on exit from the for-inspect statement is the meet of the typestate on entry to the main clause and the typestate prior to the for-inspect statement. A for-inspect statement can always terminate normally, even if the main clause terminates in an unconditional exit, since it is possible for the main clause to be executed zero times.

Size

\[
\text{secondary} ::= \text{size of secondary}
\]

The operator size of evaluates to an integer containing the number of elements in the table. The table must be initialized.

Position-of-element

\[
\text{secondary} ::= \text{position of element-variable}
\]

This operation can be applied only to constant copies of elements selected from an ordered table. These constant copies are produced only within a where expression of a selector, an inspect, or a for-inspect statement. The operation evaluates to an integer which is the position of the selected element which was used to produce the constant copy stored in the element variable. Positions begin at 0.

Example:

```
string <- "abcabd"
inspect c in string where(c = 'b')
begin
  p <- position of c;
end inspect;
```

The variable \( p \) will receive the value 1—the position of the first 'b' in the string.

Position-of-selector

\[
\text{secondary} ::= \text{position of selector}
\]

This operation searches an ordered table for the first selected element. It evaluates to an integer equal to the position of this selected element. If
there is no selected element, a NotFound exception is raised.

11.7 Variant Operations

Variant Type Family

\[
\text{type-construction} \quad ::= \text{variant of enumeration-type} \left( \quad \left[ \text{case-declaration} \mid, \text{case-declaration} \right] \ldots \right) \\
\text{case-declaration} \quad ::= \text{named-literal} \rightarrow \text{component-declaration} \quad \text{component-typestate}
\]

A variant is a value which will have one of a fixed, pre-specified set of types. Since in Hermes, every variable name has exactly one type, we use a different component name to designate each different type of value. A variant \( V \) can be either

- **uninitialized**: it has no value and there are no typestate attributes involving variable \( V \)
- **initialized and hidden**: exactly one component exists. The program doesn't know which component exists and which do not, so no component is accessible. The typestate is \( \text{init}(V) \).
- **initialized and revealed**: exactly one component has a value. The program knows which component has a value and can access that component and no other. The typestate is \( \text{init}(V), \text{case}(V, V.X) \). There may be other typestate attributes involving variable \( V.X \).

Two variants are equal if they have the same case and if their existing components are equal.

The components of a variant type are in one-to-one correspondence with the named literals of an enumeration type. These named literals are called the *cases* of the variant, and the enumeration type is the *case type*.

A variant type definition specifies: (1) the case type, (2) for each named literal of the case type, a component name, component type, and a formal *case typestate*. The case typestate is the typestate the component will have just before it is hidden and just after it is revealed. While the variant is revealed, the typestate may become higher than this typestate, but never lower.

The following is an example of a variant type definition.

\[
\text{id : charstring;} \\
\text{lisptype : enumeration ('nil', 'atom', 'pair');}
\]
s_expression: variant of listtype {
  'nil' -> null: empty {},
  'atom' -> atom: id {init},
  'pair' -> pair: cons_cell {full}
};

cons_cell: record (
  car: s_expression,
  cdr: s_expression
);

Note that these type definitions are mutually recursive—variables of type s_expression contain components of type cons_cell, which in turn contain components of type s_expression.

In this example, if V is a variable of type s_expression, and it has just been revealed to be in case 'pair', then the typestate will be

    init(V), case(V, V.Pair), init(V.Pair), init(V.Pair.Car), init(V.Pair.Cdr)

Inner components, such as V.Pair.Car.Pair.Cdr, will not be accessible until they are revealed in turn. It is not legal to discard V.Pair.Car, leaving a partially initialized pair. It is, however, legal to replace the value of V.Pair.Car.

Unite

    simple-statement
      ::= unite variant-component from source-expression

The unite statement initializes a variant to a particular one of its cases. Its two operands are a variable name designating a variant component, and an expression evaluating to a value of the same type as that variant component. The typestate of the expression must be at least as high as the case typestate of the variant component. If it is higher, it is coerced until it is exactly equal to the case typestate.

If the variant had a value, that value is discarded. The value of the expression is then moved into the variant component. After the operation, the variant will be revealed and the variant component will be accessible.

Dissolve

    simple-statement
      ::= dissolve variant-component into result-variable

The dissolve statement reverses the effect of the unite statement. The first variable name designates a variant component. The second variable name designates a variable of the same type which will receive the value of that variant component.

The variant component must be revealed. Any previous value of the destination variable is discarded. The value of the variant component is
then moved into the destination variable. After the operation, the variant will be uninitialized, and the destination variable will have the typestate attributes of the variant component.

Reveal

\[
\text{simple-statement} \quad ::= \text{reveal variant-component}
\]

The `reveal` statement reveals a hidden variant component. The operand is a variant component. The variant must be hidden—if it is revealed, it is coerced back to hidden.

If the case of the variant is such that the component being revealed exists, then that component is revealed. If the case of the variant is such that the component being revealed does not exist, the exception `CaseError` is raised, and the variant remains hidden.

Hide

\[
\text{simple-statement} \quad ::= \text{hide variant-variable}
\]

The `hide` operation is a coercion. It can be explicitly coded, or (more usually) generated automatically as a result of a coercion which drops the case attribute.

The single operand of a `hide` statement is a variable of variant type. It must be initialized. After executing the statement, the variant will be hidden.

Case

\[
\text{secondary} \quad ::= \text{case of secondary}
\]

The `case of` operator takes a variant as operand and returns the case of the variant. The result is a value of the variant's case type.
11.8 Process Creation and Communication Operations

Input Port, Output Port, Callmessage Type Families

```plaintext
type-construction
 ::= callmessage (  
    [ component-declaration [ , component-declaration ]  
    ... ]  
)  
    [ constant-parameters ]  
exit exit-typestate  
    [ minimum ]  
    [ user-exception ] ...  
 ::= import of callmessage-type entry-typestate  
 ::= output of import-type  

constant-parameters
 ::= constant ( [ component-name [ , component-name ] ...  
  ] )  
minimum
 ::= minimum minimum-typestate  
user-exception
 ::= exception user-exception-name exception-typestate
```

The type families input port, output port, and callmessage are used in "programming-in-the-large"—that is, the division of systems into processes, the creation of processes, the creation of bindings between processes, and the communication between processes.

An input port or `import` is a message queue to which connections can be made. It should be emphasized that in Hermes imports are values, and that import values are stored in import variables just as integer or string values are stored in integer or string variables. This is completely consistent with the rest of Hermes, but might appear "different" because in many languages ports or entries are not values of variables.

An import type definition specifies (1) a type name, called the message type, and (2) a formal typestate, called the entry typestate.

Usually the message type will be a callmessage type. However, the message type can be any type, since `send` can be use to transmit any value to an import.

The entry typestate is the typestate that values of the message type will have when sent. You may omit `init(*)` from the formal typestate of a callmessage, since all `call` statements will generate initialized callmessages, and using `send` to deliver an uninitialized callmessage is pointless.
Two imports compare equal only if they are the same import or constant copies of the same import. comparison of

An output port or output is a connection to an input port. As with input ports, it should be re-emphasized that output ports are treated like any other Hermes value, and can be stored in variables and passed in messages. It is the ability for programs to send and receive output ports that gives Hermes the power of capability-based systems.

Two outputs are equal when they are connections to the same input port. Each input port created with the new operation is considered a different input port. A constant copy of an input port is considered the same input port.

An output type definition specifies a matching input port type. This is the type of the input port to which the output port can be connected. The type and typestate of data sent on an output port is determined by the message type and typestate of this matching input port type.

A callmessage type definition consists of

- a set of component declarations,
- an optional set of constant parameters,
- an exit formal typestate,
- an optional minimum formal typestate, and
- an optional set of callmessage exceptions.

The component declarations define the call parameters. The order of these declarations is important, because the call arguments are matched to these parameters in the same order. The constant parameters state which components are not modifiable by the called process. The exit formal typestate specifies the typestate of the callmessage when it is returned to the caller in a normal outcome. The minimum formal typestate specifies the lowest typestate that the components of a callmessage may be lowered to. This is the typestate to which the components will be lowered when the callmessage is discarded.

The callmessage exceptions consist of an exception name and a formal typestate. The formal typestate specifies the typestate that the callmessage will have when it is returned with that exception. Do not include the Discarded exception in this list—this exception is always implicitly declared and the typestate for this exception is the same as the minimum typestate.

If there is no explicit minimum formal typestate, then it is taken to be one in which all constants are fully initialized, and all other parameters are uninitialized.

The following is an example of a callmessage and the associated ports.

```plaintext
sample_interface: callmessage (}
first: integer,
second: integer,
third: integer,
fourth: integer
)
constant (first)
exit {init(first), init(second), init(fourth)}
minimum {init(first), init(second)}
exception Failure {init(first), init(second), init(third), negative(first)};

sample_inport: inport of sample_interface
{init(first), init(second), init(third)};
sample_outport: outport of sample_inport;

In this example, parameter first is an in parameter—initialized on entry and exit, and guaranteed constant. Parameter second is an inout parameter—initialized on entry and exit, but not guaranteed constant. Parameter fourth is an out parameter—uninitialized on entry, but returned initialized on normal exit. Parameter third is a transferred parameter—owned by the caller before the call, but retained by the called process and returned uninitialized. The typestates for the exception Failure are different—parameters first, second, and third will be initialized, and parameter first will be known to be negative.

It is illegal for the entry typestate of an inport whose message type is a callmessage to specify a typestate which is not higher than or equal to the minimum typestate. Similarly all exit and exception typestate must be higher than or equal to the minimum typestate.

New

\[
simple\text{-}statement \\
::= \text{new} \ \text{variable}\text{-}name
\]

The \text{new} operation initializes an input port variable to an empty queue with no connections. The previous value of the input port variable is discarded.

Empty

\[
secondary \\
::= \text{empty of} \ \text{secondary}
\]

The \text{empty} of operation returns a true boolean value if the queue is empty, otherwise it returns false. The operand must be an initialized inport.

Since other processes may enqueue messages at arbitrary times, being empty is not a stable property; a process may repeat a successful \text{empty} of operation only to find the inport no longer empty. However, since only the process owning an input port can receive messages, being non-empty is a stable property.
Connect

\[ \text{simple-statement} \]
\[ ::= \text{connect} \ \text{outport-variable} \ \text{to} \ \text{import-variable} \]

The \text{connect} statement has two operands. The first operand (before the keyword \text{to}) is an outport variable. The second is an import variable. The import variable must be of the matching import type of the outport variable type, and must be initialized.

The previous value of the outport variable, if any, is discarded. The outport variable is assigned to a connection to the import which is the current value of the import variable.

Call

\[ \text{simple-statement} \]
\[ ::= \text{call} \ \text{outport-variable} \ \text{call-arguments} \]
\[ ::= \text{call} \ ( \ \text{outport-expression} \ ) \ \text{call-arguments} \]

\[ \text{function-reference} \]
\[ ::= \text{outport-primary} \ \text{call-arguments} \]

\[ \text{association} \]

\[ \text{associated-pair} \]

The \text{call} statement corresponds to a procedure call in conventional languages. The operands of the \text{call} statement are an outport and a list of arguments. The outport must be initialized. The matching import type must have an element type which is a callmessage. The arguments are matched up to the callmessage components either by position, if positional notation is used, or by name, if association notation is used. If function notation is used, then there should be one fewer argument than parameter, the missing parameter being the result, which is passed as an unnamed variable. If positional notation is used with a function call, then the last callmessage component is the result. If association notation is used, then the omitted component is the result.

Each argument must have a type matching its corresponding parameter. Each argument must have a typestate at least as high as the entry typestate of the corresponding parameter. The typestate of arguments matching non-constant parameters will be lowered to exactly match the entry typestate of the corresponding parameter. The typestates of arguments matching constant parameters will remain unchanged.

Overlapping variables cannot be passed as two or more arguments of a call. Two variables overlap if one of their names is a prefix of the other or they are the same.
It should be noted that expressions are evaluated into temporary variables. If these variables are passed to modifiable parameters, the modifications are lost. (For example, if you pass 3 to an inout parameter, and the procedure returns 4, you will not change the value of 3.) Since function calls are parts of expressions, and expressions may not modify their operands, no function call can have any modifiable parameter except its result.

The call is executed by: (1) creating a new callmessage value; (2) moving the arguments into the callmessage, (3) sending the callmessage to the input port to which the output port is connected, (4) waiting for the callmessage to be returned, (5) moving the callmessage components back into the argument variables, (6) throwing away the callmessage.

If several output ports are connected to the same input port, then calls on these output ports are merged fairly. This means that if a call or send is made, and the receiving process repeatedly issues receive statements, then eventually the message will be received.

If the input port has been thrown away, leaving the output port without a port connection, the Disconnected exception is raised. The arguments are returned in the same typestate they were sent.

If the call is returned with an exception, then all the above steps are carried out as in a regular call, but additionally, the exception is raised in the caller. This can only happen if there exists a declaration of that exception in the callmessage type definition. The full name for the exception includes the callmessage type and the exception name: `<cmtype>.<exceptionname>`.

If the process owning the callmessage discards it, then the exception `<cmtype>.Discarded` is raised in the caller. The callmessage is returned with the parameters in the minimum typestate as defined in the callmessage definition.

The typestate after executing a call statement is derived from the typestate just prior to the call, from the entry typestate, and from the typestate defined for the normal or exceptional return being made. The difference between the entry and return typestates defines which attributes of the parameters are added and dropped. These adds and drops are applied to the arguments matching the parameters.

For example

```
call p(a, b, c, d);
```

Suppose the type of p is `sample.outport`, defined above. Suppose the typestate prior to the call is: `{init(p), init(a), even(a), init(b), even(b), init(c)}`. Since a matches the constant parameter first, it may retain its additional attribute `even(a)`. Since b matches the non-constant parameter `second`, it cannot retain its attribute `even(b)`, which is dropped prior to the call. On normal return from the call, the typestate will be `{init(p), init(a), even(a), init(b), init(d)}`. On exception `sample.interface.Failur`, the typestate will be `{init(p), init(a), even(a), negative(a), init(b), init(c)}`.
Receive

\[ \text{simple-statement} \quad ::= \text{receive callmessage-variable from import-variable} \]

The \textit{receive} statement dequeues a message from an input port, if this is possible, and stores it in a destination variable. The input port must be initialized.

On normal completion of \textit{receive}, a message is dequeued from the input. The previous contents of the destination variable, if any, are discarded. The message is moved into the destination variable. This variable will be in the entry typestate associated with the import. The choice of which message to receive is non-deterministic.

If the import has no connections, then the \textit{Disconnected} exception will be raised.

If the import has connections, but no message is ever sent to the import, then the \textit{receive} statement will block indefinitely.

Return

\[ \text{simple-statement} \quad ::= \text{return callmessage-variable} \]

\[ \quad [ \text{exception user-exception-name} ] \]

The \textit{return} statement returns a callmessage to its caller and resumes execution of the caller. Optionally, an exception can be returned to the caller.

The \textit{return} statement must specify either no exception, or a legal exception for that callmessage type—either \textit{Discarded}, or an exception listed in the definition. The exception name is written without any type name qualifier.

The callmessage must have the typestate attributes associated with the kind of return—the exit typestate for a return without exceptions, or the appropriate exception typestate for a return with an exception. If the typestate is higher, coercions are inserted to lower the typestate to exactly that which is expected by the caller.

After the \textit{return}, the callmessage is uninitialized. The caller resumes execution. If an exception is specified, that exception is raised in the caller, not in the process issuing the \textit{return} statement. The process issuing \textit{return} continues executing.

There is no automatic termination of a process executing a \textit{return} statement, as would be the case if processes were passive procedures. To produce the equivalent effect of procedures, code the \textit{return} statement as the last executable statement of the process, or follow the \textit{return} statement with an \textit{exit} statement jumping to the end of the process.
Send

\[ \text{simple-statement} \]
\[ ::= \text{send source-expression to outport-expression} \]

The \text{send} statement moves a value out of a variable and enqueues it on an input port. Its operands are a source variable and an output.

The variable whose value is to be sent must have the correct type, and must have the entry type state or higher. The type and type state are determined from the matching input type of the output. If the type state is higher than the entry type state, it is coerced down to the entry type state.

The output port must be initialized.

If the input port to which the output port was connected has been discarded, then the \text{Disconnected} exception is raised.

Create

\[ \text{secondary} \]
\[ ::= \text{create of secondary} \]

The \text{create of} operation instantiates a new process. The operand of \text{create} is a variable of type \text{predefined program}. The result is an output port.

The program must have the type state checked.

The program value is defined in the \text{predefined} module. It consists of a set of processes. One is the \text{main program}. The others are process literals compiled within the main program or within other process literals. A main program is the result of converting the \text{process module} production of a source module.

The initialization port is the variable declared following the keyword \text{process}. This must be an input port type. When \text{create} is executed, it is checked that this input port type is the matching input port type of the output type which is the result of the \text{create} statement. If it is not, the exception \text{InterfaceMismatch} is raised.

If there is no interface mismatch, a new process is created. This process will execute the program defined by the main program in the program value, beginning at the statement following the keyword \text{begin}. Initially only the initialization port will be initialized. A connection to the initialization port is returned as the value of \text{create}.

Procedure

\[ \text{secondary} \]
\[ ::= \text{procedure of secondary} \]

The \text{procedure of} operation instantiates a process \text{generator}. The operand is a checked program, and the result is an output port. As with \text{create}, an \text{InterfaceMismatch} exception is returned if the output port type is
incompatible with the input port type of the initialization port of the new process.

The effect of procedure is identical to that of create except that what is created is not a process running the designated program, but instead is a generator process.

Each time the generator process is called, it constructs a new instance of a process running the designated program. It then sends the call message it received to the initialization port of the new process.

Ordinary processes are not recursive, since an attempt by a process to call an input port in the same process will produce a deadlock. A call to a generator process will not deadlock since each call generates a new instance of the designated program. The operation to create a generator process is called procedure of because the semantics of generator processes is exactly like the semantics of procedures in Algol-like languages.

11.9 Polymorph Operations

Polymorph Type Family

type-construction
::= polymorph

A polymorph is a value consisting of two parts: (1) a wrapped value, which can be of any type and type state, (2) a wrapper describing the type and formal type state of the wrapped value.

A polymorph type definition defines a distinct domain of polymorphs for the purposes of assignment and comparison. For example:

Resource: polymorph;
MailItem: polymorph;

Variables of type Resource and variables of type MailItem can contain wrapped values of any type; however, a variable of type Resource can not be assigned with or compared to a variable of type MailItem.

Two polymorphs can be compared for equality. The polymorphs are equal if and only if both wrappers and both wrapped values are equal.

Wrap

simple-statement
::= wrap source-expression as polymorph-variable

The wrap statement removes a value from a source variable, adds a wrapper containing the source variable's type and formal type state, and moves the resultant polymorph value into the polymorph variable. The polymorph variable itself will have a type state of init after the operation, regardless of the type state of the wrapped value. The source variable's type state will
be uninitialized. Before the `wrap` statement typestate analysis will drop any constraint which include both a variable that overlaps the source variable and a variable that does not.

Other typestate restrictions apply as for a `move` statement: neither source nor destination can be constant, and attributes other than `init` involving the destination variable will be dropped.

For example, suppose that variable `x` is of type `person`, and the typestate attributes including `x` are: `{init(x), init(x.name), init(x.id), positive(x.id), gt(x.id, y)}`. Then after executing the statement

```
wrap x as y;
```

the value of `y` will be a polymorph whose wrapped value is the original value of `x`, and whose wrapper contains a type of `person` and a formal typestate of `{init(*), init(name), init(id), positive(id)}`. The constraint `{gt(x.id, y)}` will be dropped.

The new typestate for `y` will be simply `{init(y)}`.

Unwrap

```
   simple-statement
   ::= unwrap result-variable
        from polymorph-expression unwrapped-typestate
```

The `unwrap` statement removes a wrapped value from a polymorph variable and moves the value into a destination variable. A runtime check insures that the type and typestate on the wrapper match the type and typestate expected by the destination variable. The source (polymorph) variable must be initialized.

The expected type is simply the type of the destination variable. The expected typestate is specified by coding a formal typestate within the `unwrap` statement.

If the type on the wrapper does not match the destination variable’s type, or if the typestate on the wrapper is not at least as high as the formal typestate specified on the `unwrap` statement, then a `PolymorphMismatch` exception is raised.

If the typestate on the wrapper is higher than the typestate specified on the `unwrap` statement, then the appropriate attributes are dropped before moving the value. This may entail discarding some of the wrapped value.

Other typestate restrictions apply as for a `move` statement: neither source nor destination can be constant, the source variable becomes uninitialized, attributes of the destination variable other than those defined by the formal typestate are dropped.

For example, assume that immediately after the `wrap` statement shown above, we code the statement:

```
unwrap x from y {init(*), init(id), positive(id)};
```
The types will match, but the formal typestate in the wrapper of \( y \) will include \{init(name)\}. Component \texttt{name} will be discarded before moving the wrapped value into \( x \). The resulting typestate will be \{init(x), init(x.id), positive(x.id)\}.

Inspect-polymorph

\[
\textbf{compound-statement} \\
::= \textbf{inspect} \; \texttt{declaration} \\
\texttt{from} \; \texttt{polymorph-expression} \; \texttt{inspect-typestate} \\
\texttt{begin} \\
\quad \texttt{[statement ; ...]} \\
\texttt{end inspect}
\]

The \texttt{inspect-polymorph} statement behaves like an \texttt{unwrap} statement except that instead of moving the wrapped value, a constant copy is made.

This allows the value of a polymorph to be examined, even if the polymorph is constant.

Here is an alternative version of the \texttt{unwrap} statement shown above, which would be legal even if the polymorph \( y \) were constant:

\[
\texttt{inspect x:person from y \{init(*), init(id), positive(id)\} } \\
\texttt{begin} \\
\quad \texttt{... /* statements referring to x */} \\
\texttt{end inspect;}
\]

Assuming that the wrapped value had an initialized component \texttt{name}, that component would not be discarded in the constant copy \( x \), but it would not be accessible, since the typestate inside the statement would be simply \{init(y) init(x), init(x.id), positive(x.id)\}.

The exception \texttt{PolymorphMismatch} is raised exactly as for an \texttt{unwrap} operation.

Type

\[
\texttt{secondary} \\
::= \texttt{type of secondary}
\]

The \texttt{type} of operation evaluates to the type information stored in the polymorph wrapper. This is an initialized value of type \texttt{predefined!typeof.value}.

The value includes the type name, the definitions module defining the type name, and all the imported definitions modules.

The operand must be an initialized polymorph.

Typestate

\[
\texttt{secondary} \\
::= \texttt{typestate of secondary}
\]
The typestate of operation evaluates to the typestate information stored in the polymorph wrapper. This is an initialized value of type predefined!typestateof value. The value includes the typestate, the definitions module defining the typestate, and all the imported definitions modules.

The operand must be an initialized polymorph.

11.10 Program Operations

In Hermes, you can create programs in two ways: You can write them in linear source form and pass them to the Hermes compiler for conversion into checked program values. Alternatively, you can create program values, either “from scratch”, or by composing or modifying other program values.

A program value obtained from the compiler will have the typestate attribute predefined!checked. A program built from scratch or by modifying a checked program will be missing the predefined!checked attribute. To check a fully initialized instance of program variable pgm, execute the statement:

assert checked(pgm);

In most implementations of Hermes, checking a program also entails compiling it into machine code for the machine or machines on which the program will be instantiated. This is an optional and machine-dependent optimization; the programmer cannot directly manipulate the machine code. The checking, however, is not optional. It is a typestate error to try to instantiate a program which is not checked.

The predefined!program definition is shown in the appendix.

The operations in this section are used to construct objects of type predefined!program.

Program Literal

program-literal ::= process ( declaration )

[ pragma ]
[ declaration-section ]
begin
[ statement ; ] ...
[ handler ] ...
end process ::= process link-name

A program literal operation is coded as a self-defining value. The value of the literal is the value of the program after translation by the compiler. The compiler will check the program literal at the same time that it checks
the program which embeds it. Therefore the program literal value will have the typestate attribute checked.

Example:

doubler <- create of process (y: arithQ)
declare
  init: arith;
begin
  receive init from y;
  init.result <- 2 * init.input;
  return init;
end process;

The program literal is compiled using the using and linking list of the module in which it is embedded. However, the names defined within the program literal (e.g., variables, exits) are totally separate from names defined outside the program literal. In this example, this means that there is no conflict between the variable names y and init and any occurrences of these identifiers outside the program literal. Unlike "nested procedures" of Algol-like languages, Hermes program literals cannot reference variables from the enclosing context.

Hermes provides an alternative form of the program literal so that one literal can be separately compiled and shared by many programs.

- linking( factorial )
  factorialFn <- create of process factorial;
Process factorial must have been previously compiled. Including its module-name in the linking list enables the compiler to find the compiled code and incorporate the process literal in this process.

Currentprogram

typed-primary
  ::= currentprogram

The operation currentprogram evaluates to a checked program value which when instantiated will execute the process that initially invoked currentprogram.

This operation can be used to create recursive programs:

factorial: using( arith ) process(init: arithQ)
declare
  parm: arith;
  factorial: arithFn;
begin
  receive parm from init;
  select
    where(parm.input < 0)
      return parm exception InputError;
11.10. Program Operations

where (parm.input = 0)
  parm.result <- 1;
  return parm;
where (parm.input > 0)
  factorial <- create of current program;
  parm.result <- parm.input * factorial(parm.input - 1);
  return parm;
otherwise
end select;
end process

Attributename Literal

absprog-literal
The attributename literal evaluates to a fully initialized value of type predefined!attribute.name.
For example:
  a: formal_attribute;
  ...
new a:
  a.attribute.name <- attributename init
  new a.parameters;

Typename Literal

absprog-literal
The typename literal evaluates to a fully initialized value of type predefined!typename.
It is used either in building programs, or in checking polymorph wrappers.

  d: declaration;
  ...
new d;
  d.id <- uniqueid;
  unite d.typename.typename from
typename predefined!integer
  new d.prag;

Checkdefinitions

simple-statement
::= checkdefinitions definitions-module-variable
  against definitions-library-variable
This operation completes the initialization of a definitions module. The operand is a fully initialized definitions module. The definitions module is checked for resolution errors and other restrictions (e.g. each variant definition includes a `case_id` for each possible value of the enumeration type). If the check fails, the exception `DefinitionError` is raised.

If the check succeeds, the `id` component of the definitions module is replaced by a unique identifier. All references to that identifier within the definitions module are replaced by the unique identifier. The definitions module acquires the attribute `checkeddefinitions`. The `checkeddefinitions` attribute behaves like a constraint: any attempt to modify the definitions module will discard the attribute `checkeddefinitions`. Unlike a constraint attribute, `checkeddefinitions` cannot be used in an `assert` statement; the attribute can only be asserted by executing the `checkdefinitions` statement.

Because checked definitions are given a unique module identifier, Hermes can enforce "name equivalence" of types. No other checked definitions module can possibly have the same module identifier. Therefore it is guaranteed that when two typenames or attribute names are equal, they always denote the same type or attribute definition.

### 11.11 Constraints

\[
\text{attribute-construction} \\
::= \text{constraint} ( \\
\qquad \text{[ declaration [, declaration ] ... ]} \\
\qquad \text{is constraint-typestate constraint-expression}
\]

Constraints are programmer-defined predicates on the values of variables. It is often desirable to include constraints in interfaces or in type definitions. For instance, the interface to a square-root function might require that the argument be non-negative. A program library might be defined as a table of checked programs.

In Hermes, the programmer can specify that certain constraints are to be tracked as typestate attributes. Currently, Hermes is not sophisticated enough to understand the semantics of constraint predicates well enough to track how they are preserved under all operations. For example, the compiler will not determine that adding one to a non-negative number will yield another non-negative number. What it will do is the following:

- associate a name with a predicate – e.g. `NonNegative(x)` with the predicate `x >= 0`. This defines an operation `assert NonNegative(a)` which will test whether `a >= 0` and raise an exception if it is not.

- add the typestate attribute `NonNegative(a)` if the operation `assert`
NonNegative(a) succeeds.

- track the attribute NonNegative(a) through successor statements until either a is modified, or control merges with a path in which NonNegative(a) is not in the typestate

This simple tracking of constraints is enough for many practical purposes.

Constraint Definitions

A constraint definition includes: a defining occurrence of an attribute name, a set of formal parameters, the type and typestate of these formal parameters.

Example:
NonNegative: constraint(x: integer) is
{init(x)} x >= 0;
Interval: constraint(x: integer, y: integer) is
{init(x), init(y)} x <= y;

Constraint attribute definitions occur in definitions modules and are imported exactly as are type definitions.

Assert

```
simple-statement ::= assert attribute
```

The assert operation evaluates the predicate defined by the attribute name and arguments. The programmer must supply the same number of arguments as the attribute has formal parameters. The type of each argument must match the type of the corresponding formal parameter. The typestate attributes in the attribute definition must be present at the point where the assert statement is executed. The required attributes are determined by substituting the arguments on the assert statement for the formal parameter variables in the constraint typestate.

If the predicate evaluates to 'true', then the assert statement exits normally. The typestate on normal exit will include the attribute mentioned in the assert statement. If the predicate evaluates to 'false', the ConstraintError exception is raised. If an unhandled exception occurs while evaluating the predicate, then the ConstraintFailure exception is raised.

Drop

```
simple-statement ::= drop attribute
```

The drop statement explicitly drops a constraint attribute. This operation is usually generated as a coercion when merging two paths one of which includes a constraint attribute and the other of which does not. However, it can be coded explicitly by the programmer to indicate that a constraint is about to be broken:

```plaintext
assert Invariant(Database);
while('true') repeat
    Rq <- GetTransaction();
    drop Invariant(Database);
    ... /* make changes to database */
assert Invariant(Database);
end while;
```
Appendix A

Hermes Concrete Syntax

The Hermes syntax is presented in this appendix in a format similar to the familiar BNF (Backus-Naur Form). The following notational conventions apply:

- Names of syntactic categories (nonterminal symbols) are rendered in italic.
  
  Example: `compound-statement`

- Keywords and other non-terminals (e.g. punctuation) are rendered in bold face. They should appear in programs exactly as shown (though case is insignificant).
  
  Example: `begin, <>, !`

- Optional syntax is enclosed in square brackets. The indicated construct may appear once or not at all.
  
  Example: `[ declaration-section ]`

- Repeating syntax is indicated by ellipsis. The immediately preceding construct may appear any number of times in succession. Note that if the repeated construct is not shown as optional, then at least one repetition must appear.
  
  Example: `exit-name [ , exit-name ] ...`
A.1 Lexical Rules

We begin with the rules that define the lexical tokens of the language, including identifier and various types of literal. Note that keywords and reserved words are lexically equivalent to identifiers. A table is used to distinguish them from normal identifiers. Additional logic is required to determine when a keyword is actually used as a keyword and when it is used as a normal identifier, since most keywords are not reserved. Keywords are listed in Table A.1.

Certain characters and two-character combinations constitute punctuation tokens. Punctuation sequences are listed in Table A.2.

Spaces and tabs are not allowed within tokens, except in the case of named-literal and string-literal, where they treated like any other character. Tokens may never span lines, including named literals and string literals. Arbitrary amounts of whitespace (spaces, tabs, line breaks) may appear between tokens.

There are two commenting styles:

- Two consecutive hyphens (--) begin a comment that extends to the end of the line.
- Comment text may be embedded within a line by delimiting it with the character pairs "/*" at the beginning and "*/" at the end.

A comment begins with two consecutive hyphen characters (--) and extends to the end of the line. A comment is treated as a space.

identifier ::= alpha [ alphanum ] ...  
integer-literal ::= [ sign ] digit [ digit ] ...  
real-literal ::= [ sign ] integer-part . [ fraction-part ] [ exponent ]  
integer-part ::= integer-literal  
fraction-part ::= integer-literal  
exponent ::= exponent-letter [ sign ] integer-literal  
named-literal ::= ' named-literal-character [ named-literal-character ] ... '  
string-literal ::= " [ string-literal-character ] ... "  

Character classes used in the above are as follows. Nonprinting characters other than the space character are implicitly excluded in all cases.
• *alpha* is any upper or lower case letter, or the underscore character (_).

• *digit* is any of the digits 0 through 9.

• *alphanum* includes all characters in *digit* and *digit*.

• *sign* includes the plus (+) and minus (−) signs.

• *exponent-letter* includes the upper and lower case letter 'E'.

• *named-literal-character* includes all printing characters except the single quote ('). Two consecutive single-quote characters within a *named-literal* are treated as one single quote character in this class.

• *string-literal-character* includes all characters except the double quote ("). Two consecutive double-quote characters within a *string-literal* are treated as one double quote character in this class.
<table>
<thead>
<tr>
<th>accuracy</th>
<th>drop</th>
<th>is</th>
<th>rem</th>
</tr>
</thead>
<tbody>
<tr>
<td>against</td>
<td>else †</td>
<td>keys</td>
<td>remove</td>
</tr>
<tr>
<td>and †</td>
<td>empty</td>
<td>linking</td>
<td>repeat</td>
</tr>
<tr>
<td>as</td>
<td>end †</td>
<td>merge</td>
<td>return</td>
</tr>
<tr>
<td>assert</td>
<td>enumeration</td>
<td>minimum</td>
<td>reveal</td>
</tr>
<tr>
<td>at</td>
<td>evaluate †</td>
<td>mod</td>
<td>select</td>
</tr>
<tr>
<td>attribute name</td>
<td>event †</td>
<td>moduleid</td>
<td>send</td>
</tr>
<tr>
<td>begin †</td>
<td>every</td>
<td>new</td>
<td>size</td>
</tr>
<tr>
<td>block</td>
<td>except</td>
<td>nominal</td>
<td>table</td>
</tr>
<tr>
<td>boolean</td>
<td>exception</td>
<td>not †</td>
<td>then †</td>
</tr>
<tr>
<td>call</td>
<td>exists</td>
<td>of</td>
<td>to</td>
</tr>
<tr>
<td>callmessage</td>
<td>exit</td>
<td>on †</td>
<td>true</td>
</tr>
<tr>
<td>case</td>
<td>extract</td>
<td>or</td>
<td>type</td>
</tr>
<tr>
<td>check definitions</td>
<td>false</td>
<td>ordered</td>
<td>typename</td>
</tr>
<tr>
<td>connect</td>
<td>for</td>
<td>otherwise †</td>
<td>typestate</td>
</tr>
<tr>
<td>constant</td>
<td>forall</td>
<td>output</td>
<td>unique †</td>
</tr>
<tr>
<td>constraint</td>
<td>from</td>
<td>polymorph</td>
<td>unite</td>
</tr>
<tr>
<td>convert</td>
<td>hide</td>
<td>position</td>
<td>unwrap</td>
</tr>
<tr>
<td>copy</td>
<td>if</td>
<td>pragma †</td>
<td>using</td>
</tr>
<tr>
<td>create</td>
<td>in</td>
<td>print</td>
<td>variant</td>
</tr>
<tr>
<td>current program †</td>
<td>import</td>
<td>procedure</td>
<td>where †</td>
</tr>
<tr>
<td>declare</td>
<td>insert</td>
<td>process †</td>
<td>while</td>
</tr>
<tr>
<td>definitions</td>
<td>inspect</td>
<td>real</td>
<td>wrap</td>
</tr>
<tr>
<td>discard</td>
<td>integer</td>
<td>receive</td>
<td>others †</td>
</tr>
<tr>
<td>dissolve</td>
<td>into</td>
<td>record</td>
<td></td>
</tr>
</tbody>
</table>

TABLE A.1. Hermes keywords. Reserved words are marked with a dagger (†). Reserved words cannot be used as identifiers.

| ( - various uses | ) | various uses |
| ( - typestate delimiter | ) | typestate delimiter |
| , - list separator | ; | statement terminator |
| † - module qualifier | : | is defined |
| , - component selection | <- | move |
| # - type specifier | : | concatenation |
| - > - case declaration | := | assignment |
| = - equal | <> | not equal |
| < - less than | >= | greater than |
| <= - less or equal | >= | greater or equal |
| / - divide | * | multiply |
| + - plus | - | minus |

TABLE A.2. Hermes punctuation sequences.
A.2 Syntactic Rules

A module is introduced by a header which names the module and optionally supplies lists of definitions modules to be imported and (for process modules) other process modules to be statically linked. The module body follows.

\[
\text{module} \quad ::= \quad \text{module-name} : [\text{imports}] \ [\text{linking}] \\
\text{process-module-body} \\
\quad ::= \quad \text{module-name} : [\text{imports}] \\
\text{definitions-module-body} \\
\text{imports} \quad ::= \quad \text{using} \ ( [\text{module-name} \ [\text{,] module-name }] \ldots) \\
\text{linking} \quad ::= \quad \text{linking} \ ( [\text{link-name} \ [\text{,] link-name }] \ldots)
\]

A process module body begins with a declaration for the init port, and what then follows looks almost like a block body: an optional declarations section, followed by the statements forming the process body, optionally followed by one or more exception handler specifications.

\[
\text{process-module-body} \\
\quad ::= \quad \text{process} \ (\text{declaration}) \\
\quad \quad [\text{pragma}] \\
\quad \quad [\text{declaration-section}] \\
\quad \quad \text{begin} \\
\quad \quad \quad [\text{statement;} \ldots] \\
\quad \quad \quad [\text{handler;} \ldots] \\
\quad \quad \text{end process}
\]

A declaration contains an identifier (the variable being declared) and the name of its type. A pragma is optional prior to the type name.

\[
\text{declaration-section} \\
\quad ::= \quad \text{declare} \ [\text{declaration;} \ldots]
\]

\[
\text{declaration} \quad ::= \quad \text{base-variable} : [\text{pragma}] \text{type-name}
\]

Statements come in two basic varieties: simple and compound. Either may optionally be preceded by a pragma.

\[
\text{statement} \quad ::= \quad [\text{pragma}] \text{simple-statement} \\
\quad ::= \quad [\text{pragma}] \text{compound-statement}
\]
The simple statements:

\[
\begin{align*}
\text{simple-statement} &::= \text{result-variable} \leftarrow \text{source-expression} \\
&::= \text{result-variable} := \text{source-expression} \\
&::= \text{assert} \ \text{attribute} \\
&::= \text{call} \ \text{outport-variable} \ \text{call-arguments} \\
&::= \text{call} \ (\ \text{outport-expression}\ ) \ \text{call-arguments} \\
&::= \text{checkdefinitions} \ \text{definitions-module-variable} \\
&\quad\quad\text{against} \ \text{definitions-library-variable} \\
&::= \text{connect} \ \text{outport-variable} \ \text{to} \ \text{inport-variable} \\
&::= \text{discard} \ \text{variable-name} \\
&::= \text{dissolve} \ \text{variant-component} \ \text{into} \ \text{result-variable} \\
&::= \text{drop} \ \text{attribute} \\
&::= \text{exit} \ \text{exit-name} \\
&::= \text{extract} \ \text{table-variable} \ \text{from} \ \text{selector} \\
&::= \text{hide} \ \text{variant-variable} \\
&::= \text{insert} \ \text{element-expression} \ \text{into} \ \text{table-variable} \\
&\quad\quad\text{[at position-expression]} \\
&::= \text{merge} \ \text{table-expression} \ \text{into} \ \text{table-variable} \\
&\quad\quad\text{[at position-expression]} \\
&::= \text{new} \ \text{variable-name} \\
&::= \text{print} \ \text{expression} \\
&::= \text{receive} \ \text{callmessage-variable} \ \text{from} \ \text{inport-variable} \\
&::= \text{remove} \ \text{element-variable} \ \text{from} \ \text{selector} \\
&::= \text{return} \ \text{callmessage-variable} \\
&\quad\quad\text{[exception user-exception-name]} \\
&::= \text{reveal} \ \text{variant-component} \\
&::= \text{send} \ \text{source-expression} \ \text{to} \ \text{outport-expression} \\
&::= \text{unite} \ \text{variant-component} \ \text{from} \ \text{source-expression} \\
&::= \text{unwrap} \ \text{result-variable} \\
&\quad\quad\text{from} \ \text{polyomorph-expression unwrapped-typestate} \\
&::= \text{wrap} \ \text{source-expression} \ \text{as} \ \text{polyomorph-variable}
\end{align*}
\]

The compound statements:
compound-statement
  ::= block
      [ constant-section ]
      [ declaration-section ]
      begin
      [ statement ; ] ...
      [ handler ] ...
      end block
  ::= for selector inspect
      [ statement ; ] ...
      end for
  ::= for enumeration-variable : enumeration-type repeat
      [ statement ; ] ...
      end for
  ::= if test-expression
      then [ statement ; ] ...
      [ else [ statement ; ] ... ]
      end if
  ::= inspect selector begin
      [ statement ; ] ...
      end inspect
  ::= inspect declaration
      from polymorph-expression inspect-typestate
      begin
      [ statement ; ] ...
      end inspect
  ::= select [ select-expression ]
      [ select-clause ] ...
      otherwise-clause
      end select
  ::= while test-expression repeat
      [ statement ; ] ...
      end while

Various syntactic elements appearing in compound statements:

constant-section
  ::= constant ( [ base-variable [ , base-variable ] ... ] )

handler
  ::= on ( [ exception-name [ , exception-name ] ... ] )
      [ statement ; ] ...
  ::= on exit ( [ exit-name [ , exit-name ] ... ] )
      [ statement ; ] ...

select-clause
  ::= boolean-guard [ statement ; ] ...
Appendix A. Hermes Concrete Syntax 141

 ::= event-guard [ statement ; ] ...
 ::= event-guard and boolean-guard [ statement ; ] ...

event-guard
 ::= event import-variable

boolean-guard
 ::= where ( test-expression )

otherwise-clause
 ::= otherwise [ statement ; ] ...

Three syntaxes are available for selectors (used in table operations): a long form (using where), and two shorthand forms (using square brackets — [...]).

selector
 ::= base-variable in table-variable
     where ( selector-expression )
 ::= base-variable in table-variable
     [ [ expression [ , expression ] ... ]]
 ::= table-variable [ [ expression [ , expression ] ... ]]

The Hermes expression syntax follows. All the Hermes binary operators are left-associative, though the grammar shown here does not explicitly represent this.

expression
 ::= disjunction

disjunction
 ::= conjunction
 ::= disjunction or conjunction

conjunction
 ::= relation
 ::= conjunction and relation

relation
 ::= concat
 ::= concat = concat
 ::= concat < concat
 ::= concat > concat
 ::= concat <> concat
 ::= concat <= concat
 ::= concat >= concat

concat
 ::= term
 ::= concat | term
A.2. Syntactic Rules

term ::= factor
      ::= − factor
      ::= term + factor
      ::= term − factor

factor ::= secondary
      ::= factor * secondary
      ::= factor / secondary
      ::= factor mod secondary
      ::= factor rem secondary

secondary ::= primary
            ::= the-element
            ::= case of secondary
            ::= convert of secondary
            ::= copy of secondary
            ::= create of secondary
            ::= empty of secondary
            ::= evaluate declaration from [ statement ; ] ... end
            ::= every of selector
            ::= exists of selector
            ::= forall of selector
            ::= not secondary
            ::= position of element-variable
            ::= position of selector
            ::= procedure of secondary
            ::= size of secondary
            ::= type of secondary
            ::= typestate of secondary

primary ::= type-specifier typed-primary
         ::= typed-primary
         ::= variable-name
         ::= function-reference

type-specifier ::= type-name #

typed-primary ::= literal
::= ( expression )
::= current program
::= unique

the-element
::= selector

function-reference
::= output-primary call-arguments

Literals of various types:

literal ::= integer-literal
 ::= real-literal
 ::= named-literal
 ::= string-literal
 ::= program-literal
 ::= attributename-literal
 ::= typename-literal

program-literal
 ::= process ( declaration )
::= [ pragma ]
::= [ declaration-section ]
begin
::= [ statement ; ]...
::= [ handler ]...
end process
::= process link-name

attributename-literal
 ::= \ attributename attribute-name \%

typename-literal
 ::= \ typename type-name \%

A definitions module comprises a collection of type and constraint definitions. A type definition associates a name with a type construction, while a constraint definition associates a name with an attribute construction. A pragma may optionally appear with either type of definition.
A.2. Syntactic Rules

**definition** ::= definition-name : [ pragma ] construction

**construction** ::= type-construction
                ::= attribute-construction

---

**Type constructions:**

**type-construction**

 ::= boolean ( boolean-association )
 ::= callmessage ( 
       [ component-declaration [ , component-declaration ] 
       ... ] 
 ) 
 ::= constant-parameters 
 ::= exit exit-typestate 
 ::= minimum 
 ::= user-exception ... 
 ::= [ ordered ] enumeration ( 
       [ named-literal [ , named-literal ] ... ] 
 )
 ::= import of callmessage-type entry-typestate 
 ::= integer 
 ::= nominal 
 ::= output of import-type 
 ::= polymorph 
 ::= real of accuracy integer-literal / integer-literal 
 ::= record ( 
       [ component-declaration [ , component-declaration ] 
       ... ] 
 )
 ::= [ ordered ] table of element-type element-typestate 
.toHexString 
 ::= variant of enumeration-type ( 
       [ case-declaration [ , case-declaration ] ... ] 
 )

---

Various syntactic elements used in type definitions:

**boolean-association**

 ::= true : named-literal , false : named-literal
 ::= false : named-literal , true : named-literal
constant-parameters

```plaintext
::= constant ( [ component-name [ , component-name ] ... ] )
```

minimum ::= minimum minimum-typestate

user-exception

```plaintext
::= exception user-exception-name exception-typestate
```

key ::= ( [ formal-variable [ , formal-variable ] ... ] )

case-declaration

```plaintext
::= named-literal → component-declaration component-typestate
```

component-declaration

```plaintext
::= declaration
```

Attribute constructions:

attribute-construction

```plaintext
::= constraint ( [ declaration [ , declaration ] ... ] )
```

```plaintext
is constraint-typestate constraint-expression
```

Typestate specifications appear in attribute constructions. A typestate specification consists of a collection of attributes, each of which includes an attribute name and a list of arguments. Formal typestates appear in the `unwrap` and `for ... inspect` statements, and in several type constructions (e.g., variants, tables, imports). In a formal typestate specification, each attribute argument is taken to be a component of an assumed base variable which depends on the context.

```plaintext
typestate ::= { [ attribute [ , attribute ] ... ] }
```

attribute ::= attribute-name attribute-arguments

```plaintext
formal-typestate ::= { [ formal-attribute [ , formal-attribute ] ... ] }
```

```plaintext
formal-attribute ::= attribute-name formal-attribute-arguments
```

Argument lists appear in various contexts, including `call` statements and typestate attributes. They match actual arguments to parameters. Positional matching is achieved with a comma-separated list of arguments. A keyword-based arguments list explicitly includes the parameter names so that position is unimportant.

The various types of argument lists are distinguished only by what is allowed as an argument.
call-arguments
  ::= ( [ expression , expression ] . . )
  ::= ( labeled-expression , labeled-expression ) . .

labeled-expression
  ::= identifier : expression

attribute-arguments
  ::= ( [ variable-name , variable-name ] . . )
  ::= ( labeled-variable-name , labeled-variable-name ) . .

labeled-variable-name
  ::= identifier : variable-name

formal-attribute-arguments
  ::= empty
  ::= ( [ formal-variable , formal-variable ] . . )
  ::= ( labeled-formal-variable , labeled-formal-variable ) . .

labeled-formal-variable
  ::= identifier : formal-variable

Names:

attribute-name
  ::= definition-name
  ::= module-name ! definition-name

base-variable
  ::= identifier

builtin-exception-name
  ::= identifier

component-name
  ::= identifier

definition-name
  ::= identifier

exception-name
  ::= type-name . user-exception-name
  ::= builtin-exception-name

exit-name ::= identifier
Appendix A. Hermes Concrete Syntax

formal-variable ::= *
    ::= component-name [ . component-name ]...

link-name ::= identifier

module-name ::= identifier

type-name ::= definition-name
    ::= module-name ! definition-name

user-exception-name ::= identifier

variable-name ::= base-variable [ . component-name ]...

---

Variable names appearing in particular contexts

callmessage-variable ::= variable-name

definitions-library-variable ::= variable-name

definitions-module-variable ::= variable-name

element-variable ::= variable-name

enumeration-variable ::= base-variable

import-variable ::= variable-name

outport-variable ::= variable-name

polymorph-variable ::= variable-name

result-variable ::= variable-name

table-variable ::= variable-name
A.2. Syntactic Rules

\[
\begin{align*}
\text{variant-variable} & \quad ::= \text{variable-name} \\
\text{variant-component} & \quad ::= \text{variable-name}
\end{align*}
\]

Expressions appearing in particular contexts:

\[
\begin{align*}
\text{constraint-expression} & \quad ::= \text{expression} \\
\text{element-expression} & \quad ::= \text{expression} \\
\text{outport-expression} & \quad ::= \text{expression} \\
\text{outport-primary} & \quad ::= \text{primary} \\
\text{polymorph-expression} & \quad ::= \text{expression} \\
\text{position-expression} & \quad ::= \text{expression} \\
\text{select-expression} & \quad ::= \text{expression} \\
\text{selector-expression} & \quad ::= \text{expression} \\
\text{source-expression} & \quad ::= \text{expression} \\
\text{table-expression} & \quad ::= \text{expression} \\
\text{test-expression} & \quad ::= \text{expression}
\end{align*}
\]

Types appearing in particular contexts:

\[
\begin{align*}
\text{callmessage-type} & \quad ::= \text{type-name} \\
\text{element-type} & \quad ::= \text{type-name} \\
\text{enumeration-type} & \quad ::= \text{type-name} \\
\text{import-type} & \quad ::= \text{type-name}
\end{align*}
\]
Types appearing in particular contexts:

- `component-typestate` ::= `formal-typestate`
- `constraint-typestate` ::= `typestate`
- `element-typestate` ::= `formal-typestate`
- `entry-typestate` ::= `formal-typestate`
- `exception-typestate` ::= `formal-typestate`
- `exit-typestate` ::= `formal-typestate`
- `inspect-typestate` ::= `formal-typestate`
- `minimum-typestate` ::= `formal-typestate`
- `unwrapped-typestate` ::= `formal-typestate`

Various odds and ends. Note that the pragma syntax is currently very open-ended. It may become more structured in the future.

- `pragma` ::= `pragma string-literal`
- `empty` ::=
## Appendix B

### Hermes Operations

This chapter lists all the hermes operations and provides precise rules for type checking, type inferencing, and typestate checking each operation.

### B.1 Operation Descriptions

A typical operation description appears in Figure B.1. The example will be used throughout this section.

<table>
<thead>
<tr>
<th><code>merge-at(destination, source, position)</code></th>
<th>Exceptions: Depletion, RangeError, (DuplicateKey)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type Rules:</strong></td>
<td></td>
</tr>
<tr>
<td><code>destination ∈ orderedtable</code></td>
<td></td>
</tr>
<tr>
<td><code>source ∈ table</code></td>
<td></td>
</tr>
<tr>
<td><code>position ← predefined!integer</code></td>
<td></td>
</tr>
<tr>
<td><code>destination ≡ source</code></td>
<td></td>
</tr>
<tr>
<td><strong>Preconditions:</strong></td>
<td><strong>Postconditions:</strong></td>
</tr>
<tr>
<td><code>init(source)</code></td>
<td><code>makeuninit(source)</code></td>
</tr>
<tr>
<td><code>init(destination)</code></td>
<td><code>killconstraints(destination)</code></td>
</tr>
<tr>
<td><code>init(position)</code></td>
<td></td>
</tr>
<tr>
<td><code>var(source)</code></td>
<td></td>
</tr>
<tr>
<td><code>var(destination)</code></td>
<td></td>
</tr>
<tr>
<td><code>duplicatekey?(destination)</code></td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td></td>
</tr>
<tr>
<td>Remove all table elements from <code>source</code></td>
<td></td>
</tr>
<tr>
<td>and insert them into <code>destination</code></td>
<td></td>
</tr>
<tr>
<td>so that the resulting position of the</td>
<td></td>
</tr>
<tr>
<td>first transferred element, if any, will</td>
<td></td>
</tr>
<tr>
<td>be equal to <code>position</code>. All other</td>
<td></td>
</tr>
<tr>
<td>transferred elements will follow</td>
<td></td>
</tr>
<tr>
<td>consecutively, in the same order as</td>
<td></td>
</tr>
<tr>
<td>they appeared in <code>source</code>. All</td>
<td></td>
</tr>
<tr>
<td>elements of <code>destination</code> that formerly</td>
<td></td>
</tr>
<tr>
<td>occupied positions at <code>position</code> or</td>
<td></td>
</tr>
<tr>
<td>beyond are shifted so that they appear</td>
<td></td>
</tr>
<tr>
<td>in the same relative order, and</td>
<td></td>
</tr>
<tr>
<td>following the last transferred element.</td>
<td></td>
</tr>
<tr>
<td><strong>Qualifier:</strong></td>
<td><code>absent</code></td>
</tr>
<tr>
<td><strong>See:</strong> <code>§11.6, p. 112</code></td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE B.1. Sample operation description**
B.1.1 Description Header

The operation name appears first, in bold-face, followed by an operand list. Our example describes the merge operation, which takes three operands: destination, source, and position. Some operations take a variable number of operands. In these cases, the operand list will appear as an ellipsis.

On the far right of the header line is a list of the exceptions that can be raised by the operation. Our example shows that Depletion, RangeError, and DuplicateKey may all be raised by the merge operation. The parentheses surrounding DuplicateKey indicate that in some cases it can be statically determined, based on the types of the operands, that the merge operation cannot raise DuplicateKey. The exact conditions determining this are specified later in the operation description. In this case, DuplicateKey can be raised only if the destination is a keyed table.

B.1.2 Type Rules

Below the description header are the type rules, which form the basis for both type checking (ensuring that operands are of the correct type), and type inferencing (deducing the type of undeclared operands from their context). The type rules come in two forms: class rules and inference rules.

A class rule specifies that an operand must be of a type that falls in a specified class. A class rule can never be used to infer a type, since a type class contains many types. The following class rule appears in the merge description:

\[
\text{destination} \in \text{orderedtable}
\]

A list of the type classes and their meanings appears in section B.2.

An inference rule constrains an operand to be of a particular type. That type may or may not depend on the type of another operand in the operation. An example in which no other operand appears can be found in the merge operation:

\[
\text{position} \leftarrow \text{predefined!integer}
\]

For an example involving two operands, we have the following from the insert instruction:

\[
\text{element} \leftarrow \text{elementtypeof} (\text{table})
\]

The first rule requires that the position operand be of the specific type, predefined!integer. This type of inference rule is sometimes called an "assignment rule." The second rule requires that element be of the type that is uniquely determined by applying the elementtypeof "inference function" to the type of the table operand. All inference functions and their meanings are listed in section B.3.
An inference rule can be used for type checking. It can also be used to infer the type of the left-hand operand, if either the rule is of the assignment variety, or if the type of the right-hand operand is already known.

A special notation is used for rules involving the sameas inference function, which identifies two operands that must have the same type. All such rules in an operation description are displayed as in the following example from the \texttt{merge.at} operation:

\begin{equation*}
\texttt{destination} \equiv \texttt{source}
\end{equation*}

In this case, the normal asymmetry of inference rules vanishes: the type of either operand can be inferred from the type of the other.

\subsection*{B.1.3 Preconditions}

Precondition and postcondition rules appear in side-by-side boxes below the type rules.

The precondition rules specify various conditions that must hold at the time the operation is to be executed. All preconditions must be statically verifiable by the compiler.

A precondition takes the form of a precondition function followed by a list of operands. An example from the \texttt{merge.at} operation is:

\begin{equation*}
\texttt{init(source)}
\end{equation*}

This states that the \texttt{source} operand must be initialized before the \texttt{merge.at} operation can take place.

Some “preconditions” do not check for any typestate attributes. They define conditions under which an exception is known to be possible. For example, the following precondition indicates that if the \texttt{destination} operand is a keyed table, then the operation can raise DuplicateKey:

\begin{equation*}
\texttt{duplicatekey?(destination)}
\end{equation*}

All the precondition functions and their meanings are listed in section B.4.

\subsection*{B.1.4 Postconditions}

The postcondition rules characterize the effects of a normal (non-exception) execution of the operation. Each rule is specified via a postcondition function followed by a list of operands. An example from the \texttt{merge.at} operation is:

\begin{equation*}
\texttt{makeuninit(source)}
\end{equation*}
This indicates that following a successful `merge_at` operation, the `source` operand will no longer be initialized.

The postcondition functions and their meanings are listed in section B.5.

### B.1.5 Special Rules

Absent from the `merge_at` example but present in some operator descriptions is a box labeled "Special Rules". This full-width box appears immediately following the typestate pre- and post-condition boxes, and lists type and typestate rules that could not be encoded in the rule tables. This treatment is mostly required when a rule cannot be expressed solely in terms of named operands. Thus, rules regarding data encoded in the statement qualifier (described below) appear as special rules. Also, rules about the operands to an operation without a fixed operand list (like `call` or `assert`) are specified in this manner.

### B.1.6 Operation Semantics

Following the post- and pre-condition boxes is a brief description of the semantic meaning of the operation. Below this, at the very bottom of the panel, are two boxes showing the `statement qualifier type` for this operation, and a pointer to the section and page in the reference manual that describes the operation in detail.

A statement qualifier contains all parameters of an operation other than its operands. For example, since a process begins with all its variables uninitialized (except its initialization port), it would be impossible to set an integer variable to the value "1" without having a special "set-to-one" operator or using a mechanism such as qualifiers. In Hermes, this particular situation is handled by the `integer literal` operation, which requires that an integer value be encoded in the qualifier.

The information that appears in qualifiers varies from statement to statement, so the qualifier itself is a variant. Its type is `predefined qualifier`, and you can find its definition in appendix C. The qualifier type identified in the operation description panel identifies which component of this variant type applies to this particular operation.

### B.2 Type Classes

Following are the type classes appearing in class rules:

- `integer`
- `real`
Type Classes

- boolean
- nominal
- enumeration
- variant
- table
- polymorph
- inport
- outport
- callmessage
- numeric: includes integer and real.
- orderedscalar: includes integer, real, and any enumeration defined with the ordered keyword present.
- enumerationorboolean: includes enumeration and boolean.
- variantcomponent: This is not, strictly speaking, a type class. It is really a variable class, comprising variables that are components of variables whose types are in the variant type class. Thus, z.x is in the class variantcomponent if and only if x is a variable whose type is in class variant.
- orderedtable: All ordered tables types, i.e., all types defined with the table type constructor and with the ordered keyword present.
- string: All ordered table types whose element types are enumerations. That is, a type t is in class string if and only if: (1) t is in class orderedtable; and (2) the element type of t is in class enumeration.
- newable: All input port, table, and record types, i.e., all types defined with either the inport, table, or record type constructor.
- copyable: All types other than input ports, callmessages, and types that can contain such objects. This class includes the classes integer, real, nominal, enumeration, boolean, and outport. It also includes the class polymorph although a polymorph object may wrap an uncopiable object (any attempt to copy such a polymorph would raise Uncopyable). A type in class variant, or a record type (defined with the record type constructor) is in class copyable if and only if the types of all the defined components are in copyable. A type in class table is in copyable if and only if the associated element type is in copyable.
B.3 Inference Functions

Following are the inference functions appearing in inference rules. Each inference function infers a type from its operand. An inference rule relates two operands, a source and a target, by specifying that target operand’s type must be the type inferred by the given inference function from the source operand’s type.

- \text{sameas}(\text{operand}): The target and source operands must have the same type.\footnote{In the operator description panels, we use the special notation \(\text{op}_i \equiv \text{op}_j \equiv \ldots \equiv \text{op}_k\) to denote all rules \(\text{op}_i \leftarrow \text{sameas}(\text{op}_j)\) for \(i \neq j\).}

- \text{casetypeof}(\text{operand}): The source type is in class \text{variant}. The target type is the enumeration type appearing in the source type’s definition.

- \text{matchinginportof}(\text{operand}): The source type is in class \text{outport}. The target type is the input port type appearing in the source type’s definition.

- \text{messagetypeof}(\text{operand}): The source type is either in class \text{inport} or in class \text{outport}. In the former case, the target type is the callmessage type appearing in the source type’s definition; in the latter case, if type \(ip\) is the input port type appearing in the source type’s definition, then the target type is the callmessage type appearing in the definition of \(ip\).

- \text{elementtypeof}(\text{operand}): The source type is in class \text{table}. The target type is the associated element type appearing in the source type’s definition.

B.4 Precondition Functions

B.4.1 Typestate Preconditions

The following preconditions ensure proper typestate prior to an operation. Each precondition can specify typestate attributes that are required in the program typestate before invoking the operation, as well as attributes that must not be present.

In several cases, required attributes are determined by substituting an operand into a formal typestate that is somehow associated with the operation. This substitution consists of the following: (1) replace any ‘*’ operand to any attribute in the formal typestate with the operation operand; and (2) insert the operation operand, and a dot (‘.’) before all other attribute
operands, deriving components of the operation operand. For step (1), recall that any attribute without operands is treated as an attribute applied to a single '*' operand.

As an example, we substitute $x$ into the formal typestate:

\[
\{ \text{init}, \text{case}(\ast, \ast.a) \}
\]

The resulting typestate is:

\[
\{ \text{init}(x), \text{case}(x, z.a) \}
\]

- **init(operand)**: The attribute \text{init}(\text{operand}) must be present.
- **full(operand)**: The attribute \text{init}(\text{operand}) is required. In addition, if \text{operand} is a variant, record, or callmessage, then the precondition \text{full}(\text{operand}.x) must be (recursively) satisfied for every component $x$ contained in \text{operand}.
- **casets(operand, varcomp)**: The required attributes are found by substituting \text{operand} into the case typestate of the variant component \text{varcomp} (as given in the relevant variant type definition). The forbidden attributes are all other attributes involving \text{operand} or its subcomponents.
- **initwithoutcase(varcomp)**: The attribute \text{init}(\text{variant}) is required, where \text{variant} is the variant variable of which \text{varcomp} is a component. All other attributes (notably, casets) involving \text{variant} or its subcomponents are forbidden.
- **checked(operand)**: The attributes \text{init}(\text{operand}) and \text{checked}(\text{operand}) are required. There are no forbidden attributes.
- **lowestelementstate(element, table)**: The required attributes are found by substituting \text{element} into the element typestate provided in the definition of \text{table}'s type. The forbidden attributes are all other attributes involving \text{element} or its subcomponents.
- **callpreconditions()**: This precondition appears only with the call operation. The following nonstandard substitution procedure is used:
  
  Let \text{ip} be the input port type associated with the output port being called, and let \text{cm} be the callmessage type associated with \text{ip}. The type definition for \text{ip} includes a formal typestate in which the components of \text{cm} appear as arguments. Substitute, for each such component name in this formal typestate, the corresponding actual argument in the call operation. The resulting typestate comprises the attributes required by this precondition. If there are non-constant parameters, then all attributes involving any of the corresponding actual parameters or their subcomponents, except required components as determined above, are forbidden.
Appendix B. Hermes Operations

- **lowestentrycondition**(message, port): This precondition appears only in the send operation. Let ip be the input port type associated with the port operand, which is an output port. Substitute message into the formal typestate appearing in ip's type definition to yield the required attributes. All other attributes involving message or its subcomponents.

- **lowestpostcondition**(message): This precondition appears only with the return and return-exception operations. In the former case, the normal exit typestate appearing in the callmessage type definition for message determines the preconditions; in the latter case, the typestate associated with the exception being returned determines the preconditions. In either case, message is substituted into the formal typestate in to yield the required attributes. All other attributes involving message or its components are forbidden.

- **polymorphprecondition**(operand): This precondition appears only with the wrap operation. If a formal typestate is specified in the operation, then the precondition require the attributes obtained by substituting operand into the formal typestate and forbid all other attributes involving operand or its subcomponents. If no formal typestate is given, this precondition has no effect.

- **assertable**( ): This precondition appears only with the assert operation. If the constraint being asserted is already present, then this precondition has no effect. Otherwise, substitute actual arguments for formal parameters in the typestate appearing with the constraint definition to yield the required attributes. There are no forbidden attributes.

### B.4.2 Context Preconditions

The following precondition functions test context-dependent properties of variables other than typestate.

- **var**(operand): The value of operand value may change as a result of the operation. It must not be constant in the current scope (e.g. via the constant list in a callmessage definition, or in the body of an inspect or for ... inspect statement).

- **pos**(operand): operand must be the selector variable in an "active" selector involving an ordered table. An active selector is one whose scope includes the current operation, or is associated with an inspect

---

2Note that it is impossible, with the current syntax, to specify a formal typestate in a wrap statement, so this precondition never has any effect.
or for ... inspect statement whose body includes the current operation.

B.4.3 Conditional Exceptions

The following preconditions test whether the operation is capable of raising certain exceptions in addition to its standard set.

- **rangeerror? (source, result):** This precondition appears only with the convert operation. If the domain of result does not include the conversion of every value in the domain of source, then a RangeError exception is possible.

- **duplicatekey? (table):** If the table has keys, then a DuplicateKey exception is possible.

- **uncopyable? (operand):** A type is potentially uncopiable if it is an input port or callmessage type (which are actually uncopiable), a polymorph type, a record or variant type with potentially uncopiable component types, or a table type with a potentially uncopiable element type. If operand is of a potentially uncopiable type, then the Uncopyable exception is possible.

B.5 Postcondition Functions

The following functions determine how the typestate changes on normal completion of an operation. Each function can cause attributes to be added and/or dropped.

- **makeinit (operand):** Add the attribute init(operand).

- **makefull (operand):** Add the attribute init(operand). In addition, if operand is a record or callmessage, make all its components recursively full.

- **makeuninit (operand):** Drop all attributes involving operand or its subcomponents.

- **move(s) (source, destination):** For each attribute involving source or a subcomponent: (1) drop the attribute, (2) substitute destination for source, and add the substituted attribute if not already present. Drop all other attributes involving destination variable or a subcomponent. Drop all constraint attributes of variables of which destination is a subcomponent.
- **copy**(source, destination): For each attribute involving source or a subcomponent, substitute destination for source and add the resulting attribute. Drop any other attribute involving destination or its subcomponents. Drop all constraint attributes of variables of which destination is a subcomponent.

- **makecase**(varcomp): Add the attribute case(variant, varcomp), where variant is the variable of which varcomp is a component.

- **dropcomponents**(variant): Drop any case attribute involving variant, and all attributes involving a component of variant.

- **killvariant**(varcomp): Drop all attributes involving the variant of which varcomp is a component, as well as all attributes involving its subcomponents.

- **makechecked**(operand): If operand's type is predefined!program, add the attribute checked(operand); if operand's type is predefined!definitions module, add the attribute checked definitions(operand).

- **moveelements**(table, element): Substitute element into the element formal typestate associated with the table's type, and add the resulting attributes. Drop all other attributes involving element or its subcomponents. Drop all constraint attributes involving variables of which element is a subcomponent.

- **moveentryts**(port, message): Substitute message into the entry formal typestate associated with port's (input port) type, and add the resulting attributes. Drop all other attributes involving message or its subcomponents. Drop all constraint attributes involving variables of which message is a subcomponent.

- **polymorphy**(operand): This postcondition appears only with the unwrap operation. Substitute operand into the formal typestate appearing in the unwrap statement, and add the resulting attributes. Drop all other attributes involving operand or its subcomponents. Drop all constraint attributes involving variables of which operand is a subcomponent.

- **asserted()**: This postcondition appears only with the assert operation. Add the attribute named in the assert statement.

- **killconstraints**(operand): Drop all constraint attributes involving operand or any variables of which operand is a subcomponent.
B.6 Operation Descriptions

Following are the detailed descriptions of all the Hermes operations:

<table>
<thead>
<tr>
<th>add(\text{result}, \text{source1}, \text{source2})</th>
<th>Exceptions: Depletion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type Rules:</strong></td>
<td></td>
</tr>
<tr>
<td>(\text{source1} \in \text{numeric})</td>
<td>(\text{source2} \in \text{numeric})</td>
</tr>
<tr>
<td>(\text{result} \in \text{numeric})</td>
<td>(\text{result} \equiv \text{source1} \equiv \text{source2})</td>
</tr>
<tr>
<td><strong>Preconditions:</strong></td>
<td><strong>Postconditions:</strong></td>
</tr>
<tr>
<td>(\text{init(\text{source1})})</td>
<td>(\text{makeinit(\text{result})})</td>
</tr>
<tr>
<td>(\text{init(\text{source2})})</td>
<td></td>
</tr>
<tr>
<td>(\text{var(\text{result})})</td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong> Store the sum of (\text{source1}) and (\text{source2}) in (\text{result}).</td>
<td></td>
</tr>
<tr>
<td><strong>Qualifier:</strong> absent</td>
<td>See §11.4, p. 104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>and(\text{result}, \text{source1}, \text{source2})</th>
<th>Exceptions: Depletion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type Rules:</strong></td>
<td></td>
</tr>
<tr>
<td>(\text{source1} \in \text{boolean})</td>
<td>(\text{source2} \in \text{boolean})</td>
</tr>
<tr>
<td>(\text{result} \in \text{boolean})</td>
<td>(\text{result} \equiv \text{source1} \equiv \text{source2})</td>
</tr>
<tr>
<td><strong>Preconditions:</strong></td>
<td><strong>Postconditions:</strong></td>
</tr>
<tr>
<td>(\text{init(\text{source1})})</td>
<td>(\text{makeinit(\text{result})})</td>
</tr>
<tr>
<td>(\text{init(\text{source2})})</td>
<td></td>
</tr>
<tr>
<td>(\text{var(\text{result})})</td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong> If both (\text{source1}) and (\text{source2}) are true, then set (\text{result}) to true. Otherwise set (\text{result}) to false.</td>
<td></td>
</tr>
<tr>
<td><strong>Qualifier:</strong> absent</td>
<td>See §11.4, p. 105</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>assert(...)</th>
<th>Exceptions: Depletion, ConstraintError, ConstraintFailure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type Rules:</strong> See Special Rules</td>
<td></td>
</tr>
<tr>
<td><strong>Preconditions:</strong></td>
<td><strong>Postconditions:</strong></td>
</tr>
<tr>
<td>(\text{assertable()})</td>
<td>(\text{asserted()})</td>
</tr>
<tr>
<td><strong>Special Rules:</strong> The number of operands must equal the number of parameters declared in the definition of the attribute named in the statement qualifier. Each operand must also match the type of the corresponding attribute parameter.</td>
<td></td>
</tr>
</tbody>
</table>
### assert (continued)

**Description:** Evaluate the constraint identified in the instruction qualifier with the given operands. If the constraint fails, raise ConstraintError. If it raises an exception, raise ConstraintFailure.

**Qualifier:** constraintname  
See §11.11, p. 132

### attributename(result)

**Type Rules:**

\[ \text{result} \leftarrow \text{predefined!attribute--name} \]

**Preconditions:**

\[ \text{var(result)} \]

**Postconditions:**

\[ \text{makeinit(result)} \]

**Description:** Copy the attribute name object (of type predefined!attribute--name) from the instruction qualifier into result.

**Qualifier:** attributename  
See §11.10, p. 130

### block()

**Type Rules:** None

**Preconditions:** None  
**Postconditions:** See Special Rules

**Special Rules:** The entry typestate to the main clause is identical to the entry typestate of the block statement. The entry typestate for a handler clause is the meet of the entry typestates of all exit statements that branch to the handler, along with the appropriate exception exit typestates of any statements that can branch to the handler due to an exception. The typestate on normal termination of a block statement is the meet of the normal exit typestates of all clauses (including the main clause and all handlers) that can exit normally, minus any attributes involving variables declared in the block statement.

**Description:** Execute the instructions in the scope identified by the instruction qualifier. If an exception or exit occurs that is not handled by a contained block, and if there is an applicable handler listed in this block’s qualifier, execute the handler. Note that exceptions and exits are only handled when they occur in the block’s main clause (or contained subclauses); exceptions or exits occurring within a handler are passed on to the outer containing block.

**Qualifier:** block  
See §11.3, p. 96
**call(…)**

*Exceptions*: Depletion, Disconnected

**Type Rules**: See Special Rules

**Preconditions**:
- call(preconditions())
- See also Special Rules

**Postconditions**: See Special Rules

**Special Rules**: The first operand must be in class outport, and its associated import type must have a message type in class callmessage. The remaining operands are the call arguments. The number of arguments must equal the number of components in the definition of the message type, and each argument type must be the same as the corresponding message component type. The arguments must not overlap in storage (e.g., a record may not appear in the same argument list as one of its components). The first operand must have the `init` attribute on entry. Each argument must have a type state at least as high as the entry type state specified for the corresponding parameter in the associated import type definition; arguments corresponding to non-constant parameters will be coerced down if their type state is higher than specified in the interface. The exit type state for normal or exceptional return includes `init` of the first operand, plus type states for arguments as specified in the callmessage type definition. Additionally, attributes involving only arguments corresponding to constant parameters are retained, even if they do not appear in the appropriate exit type state.

**Description**: The first operand must be an output port. Form a callmessage by moving all but the first operand into successive callmessage components, and then send the callmessage on the output port. On return of the callmessage (including exception returns), move its components back into the operand variables from which they were taken.

**Qualifier**: absent

See §11.8, p. 121

**case(result, variant)**

*Exceptions*: Depletion

**Type Rules**:
- `variant ∈ variant`
- `result ← casetypeof(variant)`

**Preconditions**:
- `init(variant)`
- `var(result)`

**Postconditions**:
- `makeinit(result)`
### Appendix B. Hermes Operations

<table>
<thead>
<tr>
<th>Description</th>
<th>Qualifier</th>
<th>Check Definitions $(\text{module, library})$</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description: Copy the enumeration value corresponding to the current case of variant into result.</td>
<td>absent</td>
<td>see §11.7, p. 117</td>
<td></td>
</tr>
<tr>
<td>Preconditions:</td>
<td></td>
<td>init(library)</td>
<td>makechecked(module)</td>
</tr>
<tr>
<td></td>
<td>full(module)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>var(module)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Description: Check that the definitions contained in module are correct and consistent, both internally and with respect to all the definitions modules in library.</td>
<td>absent</td>
<td>see §11.10, p. 130</td>
<td></td>
</tr>
<tr>
<td>Type Rules:</td>
<td></td>
<td>module $\leftarrow$ predefined!definitions--module</td>
<td></td>
</tr>
<tr>
<td></td>
<td>library $\leftarrow$ predefined!definitions--modules</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Preconditions:</td>
<td></td>
<td>init(source1)</td>
<td>makeinit(result)</td>
</tr>
<tr>
<td></td>
<td>init(source2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>var(result)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>uncopiable?(source1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>uncopiable?(source2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>duplicatekey?(result)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### concatenate (continued)

*Description:* Create a new table in `result` that contains copies of all the elements of `source1` followed by all the elements of `source2`, such that all the elements from either source table are ordered in `result` as they were in the source table.

*Qualifier:* `absent`  
See §11.6, p. 109

### connect(`outport, inport`)

*Exceptions:* Depletion

<table>
<thead>
<tr>
<th>Type Rules:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inport</code> ∈ <code>inport</code></td>
<td><code>outport</code> ∈ <code>outport</code></td>
</tr>
<tr>
<td><code>inport</code> ← <code>matchinginportof(outport)</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Preconditions:</th>
<th>Postconditions:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>init(inport)</code></td>
<td><code>makeinit(outport)</code></td>
</tr>
<tr>
<td><code>var(outport)</code></td>
<td></td>
</tr>
</tbody>
</table>

*Description:* Create an output port connected to `inport`, and store it in `outport`.

*Qualifier:* `absent`  
See §11.8, p. 121

### convert(`result, source`)

*Exceptions:* Depletion, `(RangeError)`

<table>
<thead>
<tr>
<th>Type Rules:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>source</code> ∈ <code>orderedscalar</code></td>
<td><code>result</code> ∈ <code>orderedscalar</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Preconditions:</th>
<th>Postconditions:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>init(source)</code></td>
<td><code>makeinit(result)</code></td>
</tr>
<tr>
<td><code>var(result)</code></td>
<td></td>
</tr>
<tr>
<td><code>rangeerror?(source, result)</code></td>
<td></td>
</tr>
</tbody>
</table>

*Description:* Convert the value of `source` to the appropriate corresponding value in the domain of `result`, and store the converted value in `result`.

*Qualifier:* `absent`  
See §11.4, p. 106

### copy(`result, source`)

*Exceptions:* Depletion, `(Uncopyable)`

<table>
<thead>
<tr>
<th>Type Rules:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>source</code> ∈ <code>copyable</code></td>
<td><code>result</code> ∈ <code>copyable</code></td>
</tr>
</tbody>
</table>

| `result` ≡ `source` |     |

*Qualifier:* `absent`
### Appendix B. Hermes Operations

#### copy (continued)

**Preconditions:**
- `init(source)`
- `var(result)`
- `un抄可(source)`

**Postconditions:**
- `copy(source, result)`

**Description:** Make a copy of the object stored in `source`, and store the copy in `result`.

**Qualifier:** absent 

See §11.1, p. 94

#### create(`result`, `program`)

**Exceptions:** Depletion, InterfaceMismatch

**Type Rules:**
- `result ∈ outport`
- `program ← predefined!program`

**Preconditions:**
- `checked(program)`
- `var(result)`

**Postconditions:**
- `makeinit(result)`

**Description:** Instantiate `program` as a process, and store an output port connected to the process’ initialization port in `result`.

**Qualifier:** absent 

See §11.8, p. 124

#### current_program(`result`)

**Exceptions:** Depletion

**Type Rules:**
- `result ← predefined!program`

**Preconditions:**
- `var(result)`

**Postconditions:**
- `makechecked(result)`

**Description:** Store a copy of the program object from which the executing process was instantiated, into `result`.

**Qualifier:** absent 

See §11.10, p. 129

#### discard(`variable`)

**Exceptions:** —

**Type Rules:** None

**Preconditions:**
- `var(variable)`

**Postconditions:**
- `makeuninit(variable)`
### discard (continued)

**Description:** Discard the object stored in variable, leaving variable uninitialized.

**Qualifier:** absent

See §11.1, p. 95

### dissolve(result, varcomp)

**Type Rules:**
- `varcomp ∈ variantcomponent`
- `result ≡ varcomp`

**Preconditions:**
- `cases(varcomp, varcomp)`
- `var(result)`
- `var(varcomp)`

**Postconditions:**
- `movets(varcomp, result)`
- `killvariant(varcomp)`

**Description:** Move the object stored in variant component `varcomp` into `result`, leaving the variant of which `varcomp` is a component uninitialized.

**Qualifier:** absent

See §11.7, p. 116

### divide(result, source1, source2)

**Type Rules:**
- `source1 ∈ numeric`
- `source2 ∈ numeric`
- `result ∈ numeric`
- `result ≡ source1 ≡ source2`

**Preconditions:**
- `init(source1)`
- `init(source2)`
- `var(result)`

**Postconditions:**
- `makeinit(result)`

**Description:** Store the quotient of `source1` divided by `source2` in `result`.

**Qualifier:** absent

See §11.4, p. 104

### drop(…)

**Type Rules:** See Special Rules

**Preconditions:** None

**Postconditions:** None

**Exceptions:** —
**drop** (continued)

**Special Rules:** The number of operands must equal the number of parameters declared in the definition of the attribute named in the statement qualifier. Each operand must also match the type of the corresponding attribute parameter.

**Description:** Remove the constraint identified by the instruction qualifier, applied to the given operands, from the current program typestate.

**Qualifier:** constraintname  
See §11.11, p. 132

**empty**(result, import)  
**Exceptions:** Depletion

**Type Rules:**

\[
\begin{align*}
result & \in \text{boolean} \\
import & \in \text{import}
\end{align*}
\]

**Preconditions:**

\[
\begin{align*}
\text{init}(\text{import}) \\
\text{var}(\text{result})
\end{align*}
\]

**Postconditions:**

\[
\begin{align*}
\text{makeinit}(\text{result})
\end{align*}
\]

**Description:** If there are messages queued for receipt on import, set result to false. Otherwise set result to true.

**Qualifier:** absent  
See §11.8, p. 120

**equal**(result, source1, source2)  
**Exceptions:** Depletion

**Type Rules:**

\[
\begin{align*}
result & \in \text{boolean} \\
\text{source1} & \equiv \text{source2}
\end{align*}
\]

**Preconditions:**

\[
\begin{align*}
\text{init}(\text{source1}) \\
\text{init}(\text{source2}) \\
\text{var}(\text{result})
\end{align*}
\]

**Postconditions:**

\[
\begin{align*}
\text{makeinit}(\text{result})
\end{align*}
\]

**Description:** If source1 and source2 are indistinguishable objects, set result to true. Otherwise set result to false.

**Qualifier:** absent  
See §11.1, p. 94

**every**(result, table)  
**Exceptions:** Depletion, (Uncopyable)

**Type Rules:**

\[
\begin{align*}
result & \in \text{table} \\
\text{table} & \in \text{copyable} \\
\text{result} & \equiv \text{table}
\end{align*}
\]

**See also Special Rules**
every (continued)

Preconditions:
- init(table)
- var(result)
- uncopiable?(table)

Postconditions:
- makeinit(result)

See also Special Rules

Special Rules: The statement qualifier is a selector. The result variable identified in that selector must be of type predefined!boolean and must have the init attribute on normal exit from the selector. The type of the element variable identified in the selector must be the element type of table. The entry typestate for the selector is the entry typestate of the every statement. The typestate on normal exit from the every statement is computed by applying the postcondition rules listed above to the normal exit typestate of the selector, minus any attributes involving the result or element variable. See Section ?? for a discussion of how typestates are computed for selectors.

Description: Create a new table containing a copy of every element of table that satisfies the selector identified in the instruction qualifier, and store the new table in result. If table is ordered, the elements in result appear in the same relative order as they do in table.

Qualifier: selector

exists(result, table)

Type Rules:
- result ∈ boolean
- table ∈ table

See also Special Rules

Preconditions:
- init(table)
- var(result)

Postconditions:
- makeinit(result)

See also Special Rules

Exceptions: Depletion
exists (continued)

**Special Rules:** The statement qualifier is a selector. The result variable identified in that selector must be of type `predefined!boolean` and must have the `init` attribute on normal exit from the selector. The type of the element variable identified in the selector must be the element type of `table`. The entry typestate for the selector is the entry typestate of the `forall` statement. The typestate on normal exit from the `forall` statement is computed by applying the postcondition rules listed above to the normal exit typestate of the selector, minus any attributes involving the result or element variable. See Section ?? for a discussion of how typestates are computed for selectors.

**Description:** If any element of `table` satisfies the selector identified in the instruction qualifier, then set `result` to true. Otherwise set `result` to false. In particular, if `table` is empty, set `result` to false.

**Qualifier:** selector

See §11.6, p. 110

<table>
<thead>
<tr>
<th>exit()</th>
<th>Exceptions: —</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type Rules:</strong> None</td>
<td></td>
</tr>
<tr>
<td><strong>Preconditions:</strong> None</td>
<td><strong>Postconditions:</strong> See Special Rules</td>
</tr>
<tr>
<td><strong>Special Rules:</strong> The <code>exit</code> statement cannot terminate normally and therefore has no normal termination typestate.</td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong> Raise an exit condition for the exit identified in the instruction qualifier. Control transfers to appropriate handler in the innermost containing block that handles this exit condition.</td>
<td></td>
</tr>
<tr>
<td><strong>Qualifier:</strong> exit</td>
<td>See §11.3, p. 101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>expression-block()</th>
<th>Exceptions: Depletion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type Rules:</strong> None</td>
<td></td>
</tr>
<tr>
<td><strong>Preconditions:</strong> See Special Rules</td>
<td><strong>Postconditions:</strong> See Special Rules</td>
</tr>
<tr>
<td><strong>Special Rules:</strong> The typestate on entry to the expression clause is identical to entry typestate for the <code>expression_block</code> statement. The result variable must have the <code>init</code> attribute on exit from the expression clause. The typestate on normal exit from the <code>expression_block</code> statement is the normal exit typestate of the expression clause, minus any attributes involving variables declared in the <code>expression_block</code> statement.</td>
<td></td>
</tr>
</tbody>
</table>
**expression-block (continued)**

*Description:* Execute the instructions contained in the scope identified by the instruction qualifier. Modulo exceptions, the effect will be to compute a value for a result variable also identified in the qualifier.

| Qualifier: expression block | See §11.3, p. 100 |

---

**extract(result, table)**

*Type Rules:*  
result ∈ table  
result ∈ table  
See also Special Rules

*Preconditions:*  
var(result)  
init(table)  
var(table)  
See also Special Rules

*Postconditions:*  
Postconditions: makeinit(result)  
killconstraints(table)  
See also Special Rules

*Exceptions:* Depletion

*Special Rules:* The statement qualifier is a selector. The result variable identified in that selector must be of type predefined!boolean and must have the init attribute on normal exit from the selector. The type of the element variable identified in the selector must be the element type of table. The entry typestate for the selector is the entry typestate of the extract statement. The typestate on normal exit from the extract statement is computed by applying the postcondition rules listed above to the normal exit typestate of the selector, minus any attributes involving the result or element variable. See Section ?? for a discussion of how typestates are computed for selectors.

*Description:* Remove all the elements from table that satisfy the selector identified in the instruction qualifier, assemble them into a new table, and store the new table in result. If table is ordered, the elements of result will appear in the same relative order as the did in table.

| Qualifier: selector | See §11.6, p. 111 |

---

**forall(result, table)**

*Type Rules:*  
result ∈ boolean  
result ∈ table

*Preconditions:*  
init(table)  
var(result)

*Postconditions:*  
Postconditions: makeinit(result)


### forall (continued)

**Description**: If every element of `table` satisfies the selector identified in the instruction qualifier, then set `result` to true. Otherwise set `result` to empty. In particular, if `table` is empty, set `result` to true.

**Qualifier**: `selector`  
See §11.6, p. 110

### for-enumerate()

**Type Rules**: See Special Rules

**Preconditions**: See Special Rules  
**Postconditions**: See Special Rules

**Special Rules**: The enumerator variable identified in the statement qualifier must be in class `enumeration`. The entry typestate for the body clause contains all the attributes in the meet of the entry typestate for the `for-enumerate` statement and the normal exit typestate of the body clause, as well as `init` of the enumerator variable (an interactive solution is required for this, as described in Section ??). The typestate on normal termination of the `for-enumerate` statement contains all the attributes in the normal exit typestate of the body clause, minus any attributes involving the enumerator variable or any variables declared in the body.

**Description**: Execute the statements contained in the scope identified in the instruction qualifier repeatedly. The statements are executed once for each enumeration value in the enumeration type corresponding to enumerator variable also identified in the qualifier. During each execution, the enumerator variable is set to the enumeration value corresponding to the current iteration. If the enumeration type is ordered, its values are iterated over in ascending order.

**Qualifier**: `for enumerate`  
See §11.4, p. 106

### for-inspect(table)

**Type Rules**:  
`table` is `table`  
See also Special Rules

**Preconditions**:  
`init(table)`  
See also Special Rules  
**Postconditions**: See Special Rules
for-inspect (continued)

Special Rules: The statement qualifier contains a selector. The result variable identified in that selector must be of type predefined!boolean and must have the init attribute on normal exit from the selector. The type of the element variable identified in the selector, and that of the element variable identified directly in the qualifier, must be the element type of table. The entry typestate for the selector is the entry typestate of the for_inspect statement. The entry typestate of the body clause identified in the qualifier includes init of the element variable identified in the qualifier, plus all attributes in the meet of (1) the exit typestate of the selector, minus any attributes involving the selector's result or element variable; and (2) the exit typestate of the body clause (an iterative solution is required for this, as described in Section ??). The typestate on normal termination of the for_inspect statement contains all the attributes in the normal exit typestate of the body clause, minus any attributes involving the element variable identified in the qualifier or any variable declared in the body. See Section ?? for a discussion of how typestates are computed for selectors.

Description: Execute the statements contained in the scope identified by the instruction qualifier repeatedly. The statements are executed once for each element of table that satisfies the selector identified in the qualifier. During execution of the body, the inspect variable (also identified in the qualifier) is set to a constant copy of the current matching table element. If table is ordered, matching elements will be inspected in the order they appear in table. table itself is not held constant during executions of the body.

Qualifier: inspect table

See §11.6, p. 113

greater(result, source1, source2)

Exceptions: Depletion

Type Rules:

<table>
<thead>
<tr>
<th>source1</th>
<th>source2</th>
</tr>
</thead>
<tbody>
<tr>
<td>orderedscalar</td>
<td>orderedscalar</td>
</tr>
<tr>
<td>result</td>
<td>source1</td>
</tr>
<tr>
<td>boolean</td>
<td>source1</td>
</tr>
</tbody>
</table>

Preconditions:

| init(source1) | init(source2) | var(result) |

Postconditions:

makeinit(result)
### greater (continued)

**Description:** If source1 compares greater than source2 (numerically or via an enumeration ordering), then set result to true. Otherwise set result to false.

**Qualifier:** absent

### greater-equal(result, source1, source2)

**Type Rules:**
- source1 ∈ orderedscalar
- result ∈ boolean
- source2 ∈ orderedscalar
- source1= source2

**Preconditions:**
- init(source1)
- init(source2)
- var(result)

**Postconditions:**
- makeinit(result)

**Description:** If source1 compares greater than or equal to source2 (numerically or via an enumeration ordering), then set result to true. Otherwise set result to false.

**Qualifier:** absent

### hide(variant)

**Type Rules:**
- variant ∈ variant

**Preconditions:**
- init(variant)

**Postconditions:**
- dropcomponents(variant)

**Description:** Remove typestate attributes as required by the postconditions. Runtime coercions may result, but otherwise there is no runtime effect.

**Qualifier:** absent

### if()

**Type Rules:** See Special Rules

**Preconditions:** See Special Rules

**Postconditions:** See Special Rules

**Exceptions:** —
if (continued)

Special Rules: The test variable identified in the statement qualifier must be of type predefined!boolean and must have type state init on normal exit from the test clause identified in the qualifier. The type state on entry to the test clause is the entry type state for the if statement. The type state on entry to the then clause and (if present) the else clause is identical to the exit type state from the test clause. The type state on normal exit from the if statement is the meet of the normal exit type states of the then and else clauses.

Description: Execute the statements in the test clause identified in the instruction qualifier. If the resulting value in the test variable (also identified in the qualifier) is true, then execute the statements in the "then" clause (also identified in the qualifier). Otherwise, if an "else" clause is identified in the qualifier, execute the statements in that clause.

Qualifier: if

See §11.3, p. 97

insert (table, element) Exceptions: Depletion, (DuplicateKey)

Type Rules:

\begin{align*}
\text{table} & \in \text{table} \\
\text{element} & \leftarrow \text{elementtypeof} (\text{table})
\end{align*}

Preconditions:

\begin{align*}
\text{init} (\text{table}) \\
\text{lowe} \text{stelementstate} (\text{element}, \text{table}) \\
\text{var} (\text{table}) \\
\text{var} (\text{element}) \\
\text{duplicatekey} ? (\text{table})
\end{align*}

Postconditions:

\begin{align*}
\text{makeuninit} (\text{element}) \\
\text{killconstraints} (\text{table})
\end{align*}

Description: Insert the object stored in \text{element} into \text{table}, leaving \text{element} uninitialized. If \text{table} is ordered, the new element will follow all previously existing elements.

Qualifier: absent

See §11.6, p. 110
### insert-at \((table, element, position)\)

<table>
<thead>
<tr>
<th>Exceptions: Depletion, RangeError, (DuplicateKey)</th>
</tr>
</thead>
</table>

**Type Rules:**
- \(table \in \text{ordertable}\)
- \(position \leftarrow \text{predefined!integer}\)
- \(element \leftarrow \text{elementtypeof}(table)\)

**Preconditions:**
- \(\text{init}(table)\)
- \(\text{lowestelementstate}(element, table)\)
- \(\text{init}(position)\)
- \(\text{var}(table)\)
- \(\text{var}(element)\)
- \(\text{duplicatekey?}(table)\)

**Postconditions:**
- \(\text{makeuninit}(element)\)
- \(\text{killconstraints}(table)\)

**Description:** Insert the object stored in \(element\) into \(table\), so as to make the position of the newly inserted element in the resulting table equal to \(position\), and leaving \(element\) uninitialized. All elements that previously occupied element positions at or beyond \(position\) are shifted one position higher in the table.

**Qualifier:** absent
See §11.6, p. 110

### inspect-polymorph \((polymorph)\)

<table>
<thead>
<tr>
<th>Exceptions: Depletion, PolymorphMismatch</th>
</tr>
</thead>
</table>

**Type Rules:**
- \(polymorph \in \text{polymorph}\)

**Preconditions:**
- \(\text{init}(polymorph)\)

**Postconditions:** See Special Rules

**Special Rules:** The entry typestate for the body clause identified in the statement qualifier is the entry typestate for the inspect-polymorph statement, plus all attributes resulting from substitution of the element variable identified in the qualifier into the formal typestate appearing in the qualifier. The exit typestate for the inspect-polymorph statement is the exit typestate of the body clause, minus any attributes involving the element variable or any variables declared in the body.
**inspect–polymorph (continued)**

*Description*: Execute the statements contained in the scope identified in the instruction qualifier. During execution, the inspect variable (also identified in the qualifier) is set to the value wrapped in `polymorph` and is held constant. `polymorph` itself is not held constant.

**Qualifier**: inspect polymorph

---

<table>
<thead>
<tr>
<th>inspect–table(table)</th>
<th>Exceptions: Depletion, NotFound</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type Rules</strong>:</td>
<td></td>
</tr>
<tr>
<td><code>table ∈ table</code></td>
<td></td>
</tr>
<tr>
<td>See also Special Rules</td>
<td></td>
</tr>
<tr>
<td><strong>Preconditions</strong>:</td>
<td></td>
</tr>
<tr>
<td><code>init(table)</code></td>
<td></td>
</tr>
<tr>
<td>See also Special Rules</td>
<td></td>
</tr>
<tr>
<td><strong>Postconditions</strong>:</td>
<td></td>
</tr>
<tr>
<td>See Special Rules</td>
<td></td>
</tr>
</tbody>
</table>

*Special Rules*: The statement qualifier contains a selector. The result variable identified in that selector must be of type `predefined!boolean` and must have the `init` attribute on normal exit from the selector. The type of the element variable identified in the selector, and that of the element variable identified directly in the qualifier, must be the element type of `table`. The entry typestate for the selector is the entry typestate of the `inspect-table` statement. The entry typestate of the body clause identified in the qualifier is the exit typestate of the selector, minus any attribute involving the selector’s result or element variable, plus `init` of the element variable identified directly in the qualifier. The typestate on normal termination of the `inspect-table` statement contains all the attributes in the normal exit typestate of the body clause, minus any attributes involving the element variable identified in the qualifier or any variables declared in the body. See Section ?? for a discussion of how typestates are computed for selectors.

*Description*: Execute the statements contained in the scope identified by the instruction qualifier. During execution of the body, the inspect variable (also identified in the qualifier) is set to a constant copy of a table element that satisfies the selector identified in the qualifier. If `table` is ordered, the first table element that satisfies the selector will be inspected. `table` itself is not held constant during execution of the body.

**Qualifier**: inspect table

---

See §11.9, p. 127
See §11.6, p. 112
### Appendix B. Hermes Operations

#### integer-literal(result)

**Type Rules:**
- `result ∈ integer`

**Preconditions:**
- `var(result)`

**Postconditions:**
- `makeinit(result)`

**Description:** Interpret the digit string stored in the instruction qualifier as a decimal integer, and store value in `result`.

**Qualifier:** `literal`

See §11.4, p. 104

#### less(result, source1, source2)

**Type Rules:**
- `source1 ∈ orderedscalar`
- `result ∈ boolean`
- `source2 ∈ orderedscalar`
- `source1 ≡ source2`

**Preconditions:**
- `init(source1)`
- `init(source2)`
- `var(result)`

**Postconditions:**
- `makeinit(result)`

**Description:** If `source1` compares less than `source2` (numerically or via an enumeration ordering), then set `result` to true. Otherwise set `result` to false.

**Qualifier:** `absent`

See §11.4, p. 105

#### less-equal(result, source1, source2)

**Type Rules:**
- `source1 ∈ orderedscalar`
- `result ∈ boolean`
- `source2 ∈ orderedscalar`
- `source1 ≡ source2`

**Preconditions:**
- `init(source1)`
- `init(source2)`
- `var(result)`

**Postconditions:**
- `makeinit(result)`

**Description:** If `source1` compares less than or equal to `source2` (numerically or via an enumeration ordering), then set `result` to true. Otherwise set `result` to false.

**Qualifier:** `absent`

See §11.4, p. 105
### merge\(\text{destination}, \text{source}\)  

<table>
<thead>
<tr>
<th>Exceptions: Depletion, (\text{DuplicateKey})</th>
</tr>
</thead>
</table>

**Type Rules:**
\[
\begin{align*}
\text{destination} & \in \text{table} \\
\text{source} & \in \text{table} \\
\text{destination} & \equiv \text{source}
\end{align*}
\]

**Preconditions:**
- `init(source)`
- `init(destination)`
- `var(source)`
- `var(destination)`
- `duplicatekey?(destination)`

**Postconditions:**
- `makeuninit(source)`
- `killconstraints(destination)`

**Description:** Remove all the table elements from `source` and insert them into `destination`, leaving `source` uninitialized. If the tables are ordered, then the transferred elements will appear in `destination` in the same relative order as they appeared in `source`, and following all previously existing elements of `destination`.

**Qualifier:** absent  
See §11.6, p. 112

### merge-at\(\text{destination}, \text{source}, \text{position}\)  

<table>
<thead>
<tr>
<th>Exceptions: Depletion, RangeError, (\text{DuplicateKey})</th>
</tr>
</thead>
</table>

**Type Rules:**
\[
\begin{align*}
\text{destination} & \in \text{ordertable} \\
\text{source} & \in \text{table} \\
\text{position} & \leftarrow \text{predefined!integer} \\
\text{destination} & \equiv \text{source}
\end{align*}
\]

**Preconditions:**
- `init(source)`
- `init(destination)`
- `init(position)`
- `var(source)`
- `var(destination)`
- `duplicatekey?(destination)`

**Postconditions:**
- `makeuninit(source)`
- `killconstraints(destination)`
merge-at *(continued)*

**Description:** Remove all table elements from *source* and insert them into *destination* so that the resulting position of the first transferred element, if any, will be equal to `position`. All other transferred elements will follow consecutively, in the same order as they appeared in *source*. All elements of *destination* that formerly occupied positions at `position` or beyond are shifted so that they appear in the same relative order, and following the last transferred element.

**Qualifier:** absent  
See §11.6, p. 112

<table>
<thead>
<tr>
<th><code>mod(result, source1, source2)</code></th>
<th><strong>Exceptions:</strong> Depletion, DivideByZero</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type Rules:</strong></td>
<td></td>
</tr>
<tr>
<td><code>source1 ∈ numeric</code></td>
<td><code>source2 ∈ numeric</code></td>
</tr>
<tr>
<td><code>result ∈ numeric</code></td>
<td><code>result ≡ source1 ≡ source2</code></td>
</tr>
<tr>
<td><strong>Preconditions:</strong></td>
<td></td>
</tr>
<tr>
<td><code>init(source1)</code></td>
<td></td>
</tr>
<tr>
<td><code>init(source2)</code></td>
<td></td>
</tr>
<tr>
<td><code>var(result)</code></td>
<td></td>
</tr>
<tr>
<td><strong>Postconditions:</strong></td>
<td></td>
</tr>
<tr>
<td><code>makeinit(result)</code></td>
<td></td>
</tr>
</tbody>
</table>

**Description:** Let `a` be the value of *source1*, and `b` be the value of *source2*. The value `c` having the same sign as `b`, with absolute value less than that of `b`, and satisfying the equation `a = n * b + c` for some integer `n`, is stored in *result*.

**Qualifier:** absent  
See §11.4, p. 104

<table>
<thead>
<tr>
<th><code>move(result, source)</code></th>
<th><strong>Exceptions:</strong> Depletion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type Rules:</strong></td>
<td><code>result ≡ source</code></td>
</tr>
<tr>
<td><strong>Preconditions:</strong></td>
<td><code>init(source)</code></td>
</tr>
<tr>
<td><code>var(source)</code></td>
<td><code>var(result)</code></td>
</tr>
<tr>
<td><strong>Postconditions:</strong></td>
<td><code>movets(source, result)</code></td>
</tr>
</tbody>
</table>

**Description:** Move the object stored in *source* to *result*, leaving *source* uninitialized.

**Qualifier:** absent  
See §11.1, p. 93
### multiply\((\text{result}, \text{source}1, \text{source}2)\)

**Type Rules:**
- \(\text{source}1 \in \text{numeric}\)
- \(\text{result} \in \text{numeric}\)
- \(\text{source}2 \in \text{numeric}\)
- \(\text{result} \equiv \text{source}1 \equiv \text{source}2\)

**Preconditions:**
- \(\text{init(source}1)\)
- \(\text{init(source}2)\)
- \(\text{var(result)}\)

**Postconditions:**
- \(\text{makeinit(result)}\)

**Description:** Store the product of \(\text{source}1\) and \(\text{source}2\) in \(\text{result}\).

**Qualifier:** absent

See §11.4, p. 104

### named-literal\((\text{result})\)

**Type Rules:**
- \(\text{result} \in \text{enumeration} \lor \text{boolean}\)

See also Special Rules

**Preconditions:**
- \(\text{var(result)}\)

**Postconditions:**
- \(\text{makeinit(result)}\)

**Special Rules:** The character string appearing in the statement qualifier must equal one of the names appearing in the definition of the type of \(\text{result}\).

**Description:** Set \(\text{result}\) to the enumeration value belonging to its enumeration type and named by the instruction qualifier.

**Qualifier:** literal

See §11.4, p. 104

### new\((\text{result})\)

**Type Rules:**
- \(\text{result} \in \text{newable}\)

**Preconditions:**
- \(\text{var(result)}\)

**Postconditions:**
- \(\text{makeinit(result)}\)

**Description:** Create a new object of the appropriate type and store it in \(\text{result}\). All components of a newly created record are uninitialized. A newly created table is empty. A newly created input port is not connected to any output port and has no queued messages.

**Qualifier:** absent

See §11.5, p. 107; §11.6, p. 108; and §11.8, p. 120
<table>
<thead>
<tr>
<th>Function</th>
<th>Exceptions: Depletion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>not</strong> ((result, source))</td>
<td></td>
</tr>
<tr>
<td><strong>Type Rules:</strong></td>
<td></td>
</tr>
<tr>
<td>(source \in boolean)</td>
<td>(result \in boolean)</td>
</tr>
<tr>
<td><strong>Preconditions:</strong></td>
<td></td>
</tr>
<tr>
<td>init((source))</td>
<td></td>
</tr>
<tr>
<td>var((result))</td>
<td></td>
</tr>
<tr>
<td><strong>Postconditions:</strong></td>
<td></td>
</tr>
<tr>
<td>makeinit((result))</td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td></td>
</tr>
<tr>
<td>If (source) is true, set (result) to false. Otherwise set (result) to true.</td>
<td></td>
</tr>
<tr>
<td><strong>Qualifier:</strong></td>
<td></td>
</tr>
<tr>
<td>absent</td>
<td></td>
</tr>
<tr>
<td>See §11.4, p. 105</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Exceptions: Depletion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>not-equal</strong> ((result, source1, source2))</td>
<td></td>
</tr>
<tr>
<td><strong>Type Rules:</strong></td>
<td></td>
</tr>
<tr>
<td>(result \in boolean)</td>
<td>(source1 \equiv source2)</td>
</tr>
<tr>
<td><strong>Preconditions:</strong></td>
<td></td>
</tr>
<tr>
<td>init((source1))</td>
<td></td>
</tr>
<tr>
<td>init((source2))</td>
<td></td>
</tr>
<tr>
<td>var((result))</td>
<td></td>
</tr>
<tr>
<td><strong>Postconditions:</strong></td>
<td></td>
</tr>
<tr>
<td>makeinit((result))</td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td></td>
</tr>
<tr>
<td>If (source1) and (source2) are indistinguishable objects, then set (result) to false. Otherwise set (result) to true.</td>
<td></td>
</tr>
<tr>
<td><strong>Qualifier:</strong></td>
<td></td>
</tr>
<tr>
<td>absent</td>
<td></td>
</tr>
<tr>
<td>See §11.1, p. 94</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Exceptions: Depletion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>or</strong> ((result, source1, source2))</td>
<td></td>
</tr>
<tr>
<td><strong>Type Rules:</strong></td>
<td></td>
</tr>
<tr>
<td>(source1 \in boolean)</td>
<td>(source2 \in boolean)</td>
</tr>
<tr>
<td>(result \in boolean)</td>
<td>(source1 \equiv source2)</td>
</tr>
<tr>
<td>(result \equiv source1 \equiv source2)</td>
<td></td>
</tr>
<tr>
<td><strong>Preconditions:</strong></td>
<td></td>
</tr>
<tr>
<td>init((source1))</td>
<td></td>
</tr>
<tr>
<td>init((source2))</td>
<td></td>
</tr>
<tr>
<td>var((result))</td>
<td></td>
</tr>
<tr>
<td><strong>Postconditions:</strong></td>
<td></td>
</tr>
<tr>
<td>makeinit((result))</td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td></td>
</tr>
<tr>
<td>If either (source1) is true or (source2) is true (or both are true), then set (result) to true. Otherwise set (result) to false.</td>
<td></td>
</tr>
<tr>
<td><strong>Qualifier:</strong></td>
<td></td>
</tr>
<tr>
<td>absent</td>
<td></td>
</tr>
<tr>
<td>See §11.4, p. 105</td>
<td></td>
</tr>
<tr>
<td>position-of-element(result, element)</td>
<td>Exceptions: Depletion</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td><strong>Type Rules:</strong></td>
<td></td>
</tr>
<tr>
<td>result ← predefined!integer</td>
<td></td>
</tr>
<tr>
<td><strong>Preconditions:</strong></td>
<td></td>
</tr>
<tr>
<td>pos(element)</td>
<td></td>
</tr>
<tr>
<td>var(result)</td>
<td></td>
</tr>
<tr>
<td><strong>Postconditions:</strong></td>
<td></td>
</tr>
<tr>
<td>makeinit(result)</td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td></td>
</tr>
<tr>
<td>Set result to the position occupied by element in the table on which the corresponding selector is operating. If the table has changed since the selector operation commenced, then the position of element at the time the selector operation began is used, rather than its current position.</td>
<td></td>
</tr>
<tr>
<td><strong>Qualifier:</strong></td>
<td>absent</td>
</tr>
<tr>
<td>See §11.6, p. 114</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>position-of-selector(result, table)</th>
<th>Exceptions: Depletion, NotFound</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type Rules:</strong></td>
<td></td>
</tr>
<tr>
<td>table ∈ orderedtable</td>
<td></td>
</tr>
<tr>
<td>result ← predefined!integer</td>
<td></td>
</tr>
<tr>
<td>See also Special Rules</td>
<td></td>
</tr>
<tr>
<td><strong>Preconditions:</strong></td>
<td></td>
</tr>
<tr>
<td>init(table)</td>
<td></td>
</tr>
<tr>
<td>var(result)</td>
<td></td>
</tr>
<tr>
<td>See also Special Rules</td>
<td></td>
</tr>
<tr>
<td><strong>Postconditions:</strong></td>
<td></td>
</tr>
<tr>
<td>makeinit(result)</td>
<td></td>
</tr>
<tr>
<td>See also Special Rules</td>
<td></td>
</tr>
<tr>
<td><strong>Special Rules:</strong></td>
<td></td>
</tr>
<tr>
<td>The statement qualifier is a selector. The result variable identified in that selector must be of type predefined!boolean and must have the init attribute on normal exit from the selector. The type of the element variable identified in the selector must be the element type of table. The entry typestate for the selector is the entry typestate of the position-of-selector statement. The typestate on normal exit from the position-of-selector statement is computed by applying the postcondition rules listed above to the normal exit typestate of the selector, minus any attributes involving the result or element variable. See Section ?? for a discussion of how typestates are computed for selectors.</td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td></td>
</tr>
<tr>
<td>Set result to the position of an element from table that satisfies the selector identified in the instruction qualifier. If table is ordered, use the position of the first such element.</td>
<td></td>
</tr>
<tr>
<td><strong>Qualifier:</strong></td>
<td>selector</td>
</tr>
<tr>
<td>See §11.6, p. 114</td>
<td></td>
</tr>
</tbody>
</table>
### `print(variable)`

**Type Rules:** None

**Preconditions:** None  
**Postconditions:** None

**Description:** Produce a printed representation of the object stored in `variable`. This operation, and the corresponding `print` statement in the concrete syntax, are provided only for purposes of Hermes compiler debugging, and are not meant for use by application programs. They may become unavailable without notice.

**Qualifier:** absent

---

### `procedure(outport, program)`

**Type Rules:**

- `outport` ∈ `outport`
- `program` ← `predefined!program`

**Preconditions:**

- `checked(program)`
- `var(outport)`

**Postconditions:**

- `makeinit(outport)`

**Description:** Create a procedure that will repeatedly instantiate `program` into processes, and store an output port matching the type of `program`'s initialization port in `outport`. Whenever a message is sent on this output port, a new process will be instantiated and the message forwarded to the new process' initialization port.

**Qualifier:** absent

---

### `program-literal(program)`

**Type Rules:**

- `program` ← `predefined!program`

**Preconditions:**

- `var(program)`

**Postconditions:**

- `makechecked(program)`

**Description:** Make a copy of the program object identified in the instruction qualifier and store it in `program`.

**Qualifier:** `program literal`
### real-literal(result)

**Type Rules:**
\[
\text{result} \in \text{integer}
\]

**Preconditions:**
\[
\text{var(result)}
\]

**Postconditions:**
\[
\text{makeinit(result)}
\]

**Description:** Interpret the character string stored in the instruction qualifier as a real number, and store the value in `result`.

**Qualifier:** literal

See §11.4, p. 104

### rem(result, source1, source2)

**Type Rules:**
\[
\begin{align*}
\text{source1} & \in \text{numeric} \\
\text{source2} & \in \text{numeric} \\
\text{result} & \in \text{numeric} \\
\text{result} & = \text{source1} - \text{source2}
\end{align*}
\]

**Preconditions:**
\[
\begin{align*}
\text{init(source1)} \\
\text{init(source2)} \\
\text{var(result)}
\end{align*}
\]

**Postconditions:**
\[
\text{makeinit(result)}
\]

**Description:** Let `a` be the value of `source1`, and `b` be the value of `source2`. The value `c` having the same sign as `a`, with absolute value less than that of `b`, and satisfying the equation `a = n * b + c` for some integer `n`, is stored in `result`.

**Qualifier:** absent

See §11.4, p. 104

### receive(message, import)

**Type Rules:**
\[
\begin{align*}
\text{import} & \in \text{import} \\
\text{message} & \leftarrow \text{messagetypeof(import)}
\end{align*}
\]

**Preconditions:**
\[
\begin{align*}
\text{init(import)} \\
\text{var(import)} \\
\text{var(message)}
\end{align*}
\]

**Postconditions:**
\[
\text{moveentryts(import, message)}
\]

**Description:** Dequeue the first available message from `import` and store it in `message`. If no message is available, wait for one to arrive.

**Qualifier:** absent

See §11.8, p. 123
### remove(element, table)

**Exceptions:** Depletion, NotFound

**Type Rules:**
- \(table \in table\)
- \(element \leftarrow \text{elementtypeof}(table)\)

See also **Special Rules**

**Preconditions:**
- \(\text{init}(table)\)
- \(\text{var}(table)\)
- \(\text{var}(element)\)

See also **Special Rules**

**Postconditions:**
- \(\text{moveelements}(table, element)\)
- \(\text{killconstraints}(table)\)

See also **Special Rules**

**Special Rules:** The statement qualifier is a selector. The result variable identified in that selector must be of type `predefined!boolean` and must have the `init` attribute on normal exit from the selector. The type of the element variable identified in the selector must be the element type of `table`. The entry typestate for the selector is the entry typestate of the `remove` statement. The typestate on normal exit from the `remove` statement is computed by applying the postcondition rules listed above to the normal exit typestate of the selector, minus any attributes involving the result or element variable. See Section ?? for a discussion of how typestates are computed for selectors.

**Description:** Remove an element from `table` that satisfies the selector identified in the instruction qualifier, and store it in `element`. If `table` is ordered, remove the first such element.

**Qualifier:** selector

See §11.6, p. 111

### return(callmessage)

**Exceptions:** Depletion

**Type Rules:**
- \(\text{callmessage} \in \text{callmessage}\)

**Preconditions:**
- \(\text{lowestpostcondition}(\text{callmessage})\)
- \(\text{var}(\text{callmessage})\)

**Postconditions:**
- \(\text{makeuninit}(\text{callmessage})\)

**Description:** Return `callmessage` to the process that originally sent it, without raising a user exception.

**Qualifier:** absent

See §11.8, p. 123
### return-exception(callmessage)

**Type Rules:**
\[
callmessage \in callmessage
\]

**Preconditions:**
- lowestpostcondition(callmessage)
- \(\text{makeuninit}(\text{callmessage})\)
- \(\text{var}(\text{callmessage})\)

**Postconditions:**

**Description:** Return `callmessage` to the process that originally sent it, raising the user exception identified by the instruction qualifier in that process.

**Qualifier:** `return exception`  
See §11.8, p. 123

### reveal(varcomp)

**Type Rules:**
\[
\text{varcomp} \in \text{variantcomponent}
\]

**Preconditions:**
- \(\text{initwithoutcase}(\text{varcomp})\)

**Postconditions:**
- \(\text{makecase}(\text{varcomp})\)

**Description:** Add typestate attributes as required by the postconditions. There is no runtime effect.

**Qualifier:** `absent`  
See §11.7, p. 117

### select()

**Type Rules:** See *Special Rules*

**Preconditions:** None

**Postconditions:** None

**Exceptions:** Depletion, Disconnected
select (continued)

Special Rules: All the variables identified in the event guards associated with the clauses listed in the statement qualifier must be in class import. If there is an operand, its type must be the same as that of the result variable identified in each boolean guard associated with a clause listed in the qualifier. Otherwise, each boolean guard result variable must be of type predefined!boolean. All variables associated with event guards must have the init attribute on entry to the select statement. All boolean guard result variables must have the init attribute on normal exit from their associated test clauses. The entry type state for each boolean guard clause is identical to the entry type state for the select statement. The entry type state for each select clause, including the otherwise clause, is the meet of the normal exit type state from all the boolean guard test clauses (or the select statement entry type state if there are no boolean guards). The normal exit type state from the select statement is the meet of the normal exit type states of all the select clauses, including the otherwise clause.

Description: The instruction qualifier contains a table of select clauses, each of which specifies one or both of an event guard and a boolean guard, as well as an associated statement clause. An event guard is an input port variable, while a boolean guard identifies a variable and a statement clause. The select operation first executes all the statements in clauses identified by boolean guards, yielding values for all the boolean guard variables. If the select operation has an operand, it is tested for equality with each boolean guard variable, and each select clause for which the test succeeds becomes “enabled.” If the select operation has no operands, the boolean guard values must be boolean, and select clauses with true boolean guard values are enabled. In addition, select clauses without boolean guards are enabled. If any select clause is enabled, one enabled clause with either no event guard, or with an event guard whose associated input port is nonempty, is selected. The statements in the statement clause associated with the selected clause are then executed. If there are no enabled select clauses, then the statements in the “otherwise” clause (identified in the instruction qualifier) are executed.

Qualifier: select

send(outport, message) Exceptions: Depletion, Disconnected

Type Rules:

\[
\begin{align*}
\text{outport} & \in \text{outport} \\
\text{message} & \leftarrow \text{messagetypeof(outport)}
\end{align*}
\]
| send (continued)                                      |  
|---|---|
| **Preconditions:**  
| lowestentrycondition(message, outport)  
| init(outport)  
| var(message)  
| **Postconditions:**  
| makeinit(message)  
| var(message)  
| **Description:** Add message to the queue associated with the input port connected to outport, leaving message uninitialized.  
| **Qualifier:** absent  
| See §11.8, p. 124  
|  
| size(result, table)  
| **Type Rules:**  
| table ∈ table  
| result ∈ predefined!integer  
| **Preconditions:**  
| init(table)  
| var(result)  
| **Postconditions:**  
| makeinit(result)  
| **Description:** Set result equal to the number of elements in table.  
| **Qualifier:** absent  
| See §11.6, p. 114  
|  
| string-literal(result)  
| **Type Rules:**  
| result ∈ string  
| See also Special Rules  
| **Preconditions:**  
| var(result)  
| duplicatekey?(result)  
| **Postconditions:**  
| makeinit(result)  
| **Special Rules:** Each element of the table appearing in the statement qualifier must correspond to one of the enumeration values appearing in the definition of the element type of result. That is, a singleton table containing the element must equal one of the enumeration values.  
| **Description:** Copy the string stored in the instruction qualifier and store the copy in result.  
| **Qualifier:** literal  
| See §11.6, p. 109  
|
### Subtract ($result, source1, source2$)

**Exceptions:** Depletion

<table>
<thead>
<tr>
<th>Type Rules:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$source1 \in \text{numeric}$</td>
<td>$source2 \in \text{numeric}$</td>
</tr>
<tr>
<td>$result \in \text{numeric}$</td>
<td>$result \equiv source1 \equiv source2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Preconditions:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>init($source1$)</td>
<td></td>
</tr>
<tr>
<td>init($source2$)</td>
<td></td>
</tr>
<tr>
<td>var($result$)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Postconditions:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>makeinit($result$)</td>
<td></td>
</tr>
</tbody>
</table>

**Description:** Subtract $source2$ from $source1$ and store the result in $result$.

**Qualifier:** absent

See §11.4, p. 104

---

### The-element ($result, table$)

**Exceptions:** Depletion, NotFound, (Uncopyable)

<table>
<thead>
<tr>
<th>Type Rules:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$table \in \text{table}$</td>
<td>$table \in \text{copyable}$</td>
</tr>
<tr>
<td>$result \leftarrow \text{elementtypeof}(table)$</td>
<td></td>
</tr>
</tbody>
</table>

See also Special Rules

<table>
<thead>
<tr>
<th>Preconditions:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>init($table$)</td>
<td></td>
</tr>
<tr>
<td>var($result$)</td>
<td></td>
</tr>
<tr>
<td>uncopiable?($table$)</td>
<td></td>
</tr>
</tbody>
</table>

See also Special Rules

<table>
<thead>
<tr>
<th>Postconditions:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>moveelementss($table, result$)</td>
<td></td>
</tr>
</tbody>
</table>

**Special Rules:** The statement qualifier is a selector. The result variable identified in that selector must be of type predefined!boolean and must have the init attribute on normal exit from the selector. The type of the element variable identified in the selector must be the element type of $table$. The entry typestate for the selector is the entry typestate of the the-element statement. The typestate on normal exit from the the-element statement is computed by applying the post-condition rules listed above to the normal exit typestate of the selector, minus any attributes involving the result and element variable. See Section ?? for a discussion of how typestates are computed for selectors.

**Description:** Copy an element from $table$ that satisfies the selector identified in the instruction qualifier, and store the copy in $result$.

**Qualifier:** selector

See §11.6, p. 109
### type(result, polymorph)

**Type Rules:**
- `polymorph` ∈ `polymorph`
- `result` ← `predefined!typeof--value`

**Preconditions:**
- `init(polymorph)`
- `var(result)`

**Postconditions:**
- `makefull(result)`

**Description:** Store the type of the object wrapped inside `polymorph` in `result`, along with any definition modules necessary to resolve the type.

**Qualifier:** absent

See §11.9, p. 127

### typename(result)

**Type Rules:**
- `result` ← `predefined!typename`

**Preconditions:**
- `var(result)`

**Postconditions:**
- `makefull(result)`

**Description:** Make a copy of the typename found in the instruction qualifier, and store it in `result`.

**Qualifier:** typename

See §11.10, p. 130

### typestate(result, polymorph)

**Type Rules:**
- `polymorph` ∈ `polymorph`
- `result` ← `predefined!typestateof--value`

**Preconditions:**
- `init(polymorph)`
- `var(result)`

**Postconditions:**
- `makeinit(result)`

**Description:** Store the typestate of the object wrapped inside `polymorph` in `result`, along with any definition modules necessary to resolve constraint attributes.

**Qualifier:** absent

See §11.9, p. 127
### unary-minus\((result, source)\)

**Type Rules:**
- \(result \in \text{numeric}\)
- \(result \equiv source\)

**Preconditions:**
- init\(\langle source \rangle\)
- var\(\langle result \rangle\)

**Postconditions:**
- makeinit\(\langle result \rangle\)

**Description:** Negate \(source\) and store the result in \(result\).

**Qualifier:** absent

See §11.4, p. 104

### unique\((result)\)

**Type Rules:**
- \(result \in \text{nominal}\)

**Preconditions:**
- var\(\langle result \rangle\)

**Postconditions:**
- makeinit\(\langle result \rangle\)

**Description:** Create a new nominal value, distinguishable from all other nominal values that have ever been created, and store it in \(result\).

**Qualifier:** absent

See §11.4, p. 105

### unite\((varcomp, source)\)

**Type Rules:**
- \(varcomp \in \text{variant component}\)
- \(varcomp \equiv source\)

**Preconditions:**
- case\(\langle source, varcomp \rangle\)
- var\(\langle varcomp \rangle\)
- var\(\langle source \rangle\)

**Postconditions:**
- mov\(\langle source, varcomp \rangle\)
- makecase\(\langle varcomp \rangle\)

**Description:** Put the variant of which \(varcomp\) is a component into whatever state corresponds to \(varcomp\), and then move the value stored in \(source\) to \(varcomp\), leaving \(source\) uninitialized.

**Qualifier:** absent

See §11.7, p. 116
**unwrap**(result, polymorph)  

*Type Rules:*  
polymorph ∈ polymorph

*Preconditions:*  
init(polymorph)  
var(result)

*Postconditions:*  
polyorphs(result)  
makeuninit(polymorph)

*Description:* Move the value wrapped inside polymorph to result, leaving polymorph uninitialized and coercing result as required to lower its typestate to that given by the instruction qualifier.

*Qualifier:* wrap  
See §11.9, p. 126

**while()**  

*Type Rules:* None

*Preconditions:* None  
*Postconditions:* See Special Rules

*Special Rules:* The result variable identified in the statement qualifier must be of type predefined!boolean and must have the init attribute on normal exit from the test clause identified in the qualifier. The typestate on entry to the test clause is the meet of the entry typestate of the while statement and the typestate on normal termination of the repeated clause. The typestate on entry to the repeated clause, and the normal exit typestate for the while statement, are identical to the normal exit typestate of the test clause (an iterative solution is required for this, as described in Section ??).

*Description:* Repeatedly execute the statements in the clause identified in the instruction qualifier. Prior to each iteration, execute the statements in the test clause (also identified in the qualifier) to yield a value for the test result variable (identified in the qualifier). If the test result is ever false, terminate execution of the while operation immediately (without performing the current iteration).

*Qualifier:* while  
See §11.3, p. 97

**wrap**(polymorph, source)  

*Type Rules:*  
polymorph ∈ polymorph

*Exceptions:* Depletion
**wrap (continued)**

<table>
<thead>
<tr>
<th>Preconditions:</th>
<th>Postconditions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>polymorph precondition(source)</td>
<td>makeuninit(source)</td>
</tr>
<tr>
<td>var(polymorph)</td>
<td>makeinit(polymorph)</td>
</tr>
<tr>
<td>var(source)</td>
<td></td>
</tr>
</tbody>
</table>

**Description:** Coerce `source` as necessary to lower its typestate to that given in the instruction qualifier. Then wrap it up with its type and typestate and store the result in `polymorph`, leaving `source` uninitialized.

**Qualifier:** `wrap`  
See §11.9, p. 125
Appendix C

Predefined Module

This appendix provides a complete listing of the predefined.d definitions module. The predefined module is implicitly imported by all Hermes modules. It contains all the type definitions required by the compiler to enforce assignment-style type rules. For example, the operation description for the insert-at operation requires that the position argument be of type predefined!integer. Thus there must be a type named integer defined in a definitions module named predefined in order for the compiler to function. Many types are defined in predefined because the module must be completely self-contained. For example, type program is included because it is referenced in the type rules for create and procedure. Since the definition of program makes use of predefined!processid, a definition for the latter is also included.

It should be stressed that the predefined module is not intended to provide a collection of type definitions that people will find generally useful in their programming. The only criterion by which a type definition is chosen for inclusion is the one presented above. It is not uncommon to write a Hermes application that makes no explicit use of anything from predefined.

The vast majority of the module centers around the type predefined!program, which is used to represent abstract Hermes programs. Any program that intends to manipulate Hermes program objects as data will necessarily depend heavily on the definitions in predefined.
predefined: using ();
definitions

---

---Fundamental Types---

empty: enumeration ();

option: enumeration ('present', 'absent');

integer: integer;

boolean: boolean (true: 'true', false: 'false');


charstring: ordered table of char { init };
program pragma "program" record (  
definitions_modules: definitions_modules, -- imported modules  
main_program: processid, -- the main program  
programs: processes -- one or more programs  
-- there can be multiple programs because of program literals  
);  
definitions_modules: table of definitions_module  
  { full /*, checkeddefinitions */ } keys (id);  
definitions_module: record (  
id: moduleid, -- unique id of module  
type_definitions: type_definitions, -- type definitions  
attr_definitions: attr_definitions -- attribute definitions  
);  

--- Names and IDs ---  

processid: nominal; -- identifies a process object  
clauseid: nominal; -- clause identifiers  
rootid: nominal; -- root name identifiers  
typeid: nominal; -- type identifiers  
moduleid: nominal; -- module id  
scopeid: nominal; -- scope identifier  
componentid: nominal; -- component identifier  
exceptionid: nominal; -- exception id  
statementid: nominal; -- statement identifier  
attributeid: nominal; -- attribute identifier  
exid: nominal; -- id of exit handler  

-- The following is to allow, in absprog, variables with undeclared  
-- types. For example, temporary variables that get introduced when expanding  
-- expressions to assignment statements. Their types are not known  
-- until the type checking phase which comes after resolution.  

typename_option: enumeration ( 'named', 'unnamed');  

optional_typename: variant of typename_option (  
  'unnamed' -> noname: empty { },
Appendix C. Predefined Module  197

'named' -> typename: typename { full } 
};

typename: record (  
    moduleid: moduleid,  -- time stamp of module
    typeid: typeid  -- type id within module
);  

constraintname: record (  
    moduleid: moduleid,  -- time stamp of module
    attributeid: attributeid  -- attribute id within module
);  

-- an object (variable) gives (i) the scope where it is declared, (ii)  
-- its rootid, and (iii) if it is a component, its component id.  
-- e.g., if in scope s there is the declaration r: Q, where Q is a  
-- record having a component z, then the object r.z gets would have  
-- the objectname (s, r, z).

rootname: record (  
    scope: scopeid,  -- scope of root declaration
    root: rootid  -- root identifier used to
                  -- refer to this object
);  

objectname: record (  
    root: rootname,  -- root object
    components: componentlist  -- identifiers of components
);  

componentlist: ordered table of componentid { full } ;

objectnames: ordered table of objectname { full } ;

--------------------------------------------------------------------------

Type Definitions

--------------------------------------------------------------------------

type definitions: table of type definition { full } keys (id);  

-- Each type has a unique (within its definitions module) type id, a
--- set of typed components (empty except for records, variants and 
callmessages), a specification consisting of a primitive type and 
associated information, and a pragma string

```
typedefinition: record (  
id: typeid,  
component_declarations: component_declarations,  
specification: specification_type,  
prag: charstring  
);

component_declarations: ordered table of component_declaration { full }  
  keys (id);

component_declaration: record (  
id: componentid,  
type: typename  
);

primitive_types: ordered enumeration (  
 'nominaltype', 'integertype', 'booleantype', 'enumerationtype', 'realltype',  
 'recordtype', 'varianttype', 'tabletype',  
 'inporttype', 'outporttype', 'callmessagetype',  
 'polymorphype' );

specification_type: variant of primitive_types(  
 'nominaltype' -> nominal_info: empty { },  
 'integertype' -> integer_info: empty { },  
 'booleantype' -> boolean: boolean_info { full },  
 'enumerationtype' -> enumeration: enumeration_info { full },  
 'realltype' -> accuracy_info: accuracy_info { full },
  
 'recordtype' -> record_info: empty { },  
 'varianttype' -> variant_info: variant_info { full },  
 'tabletype' -> table_info: table_info { full },  
  
 'inporttype' -> import_info: import_info { full },  
 'outporttype' -> outport_info: typename { full },  
 'callmessagetype' -> callmessage_info: callmessage_info { full },
  
 'polymorphype' -> polymorph_info: empty { })
```
Appendix C. Predefined Module

-- Specific information needed to completely specify a type falling in
-- each of the various primitive type categories

boolean_info: record (
    true_name: charstring,
    false_name: charstring
);

enumeration_info: record (
    ordered: boolean, -- is it an ordered enumeration?
    values: enumeration_values -- components
);

enumeration_values: ordered table of charstring { init } keys (*);

accuracy_info: record ( -- for real types
    accuracy_numerator: integer,
    accuracy_denominator: integer
);

variant_info: record ( -- the enumeration tagging the variant
    case_type: typename,
    case_mapping: partitionset -- maps case tags (from case-type)
        { enumeration) to components
);

partitionset: table of partition_info { full } keys (component_id);

partition_info: record ( -- component of variant
    component_id: componentid,
    case_id: integer, -- corresponding enumeration value
    case_typestate: formal_typestate -- typestate of component
);

table_info: record ( -- is the table ordered?
    ordered_table: boolean,
    keys: keyset, -- all the keys
    element_type: typename, -- type of each table element
    element_typestate: formal_typestate -- typestate of each table element
);
keyset: ordered table of formal_objects { full } keys (*);

-- Each key consists of a set of (sub-)components of the element type.
-- An empty set indicates the entire element object

formal_objects: ordered table of component_list { full };

import_info: record (  
   message_type: typename,    -- type of each queued message
   message_typestate: formal_typestate -- typestate of each queued message
);

callmessage_info: record (  
   constants: component_set,    -- which components are held constant
   normal: formal_typestate,    -- exit typestate for normal return
   -- following identifies the Discarded exception for this
   -- callmessage. The associated exit typestate is the minimum
   -- typestate for the callmessage type.
   minimum: exceptionid,
   -- following associates exit typestates with exceptions
   exception_specifications: exception_specifications
);

component_set: table of componentid { init } keys (*);

exception_specifications: table of exception { full } keys (exceptionid);

exception: record (  
   exceptionid: exceptionid,    -- unique name of exception
   post_typestate: formal_typestate -- typestate on that outcome
);

Attribute Definitions

attr_definitions: table of attr_definition { full } keys (attributeid);

-- An attribute definition includes a unique (within a definitions
-- module) identifier, the code that evaluates the attribute,
-- parameters, and the required entry typestate of the parameters.
-- The parameters exist in the outer scope of the code.
attr_definition: record (
    attributeid: attributeid, -- unique id of attribute
    execution_environment: execution_environment, -- expression to evaluate
    parameters: rootids, -- declared in the outer scope
    returnvalue: objectname, -- boolean object
    pretypestate: typestate, -- typestate on entry
    prag: charstring
);  
rootids: ordered table of rootid { init };  


--- Process Modules ---

processes: table of proc { full } keys (id);

-- Each process has a unique identifier. The code is an object of
-- type execution_environment, which includes the statements appearing
-- in the code, as well as declarations for all the root objects
-- introduced by included scopes. The initport is always an object in
-- the outermost scope.

proc: record (
    id: processid,
    executable_part: execution_environment, -- dcls + statements
    initport: rootid, -- name of initialization port
    prag: charstring
);  

-- Following record represents a piece of code, either from a process
-- module or from an attribute definition. The code is represented as
-- a collection of nested scopes, with a single main, or outermost
-- scope. Each scope introduces new declarations for root objects,
-- and is associated with a clause. A clause is a list of statements,
-- some of which introduce nested scopes. Execution of the code
-- always begins with the clause associated with the outermost scope.

execution_environment: record (
    scopes: scopes,
    }
clauses: clauses,
main_scope: scopeid
);

scopes: table of scope { full } keys (id);

scope: record (
  id: scopeid, -- scope identifier
  declarations: declarations, -- declarations introduced by scope
  clause: clauseid -- statements to execute in scope
);

-- Each declaration identifies the declared root object, and
-- optionally associates a type with the object. If no type is
-- supplied in a declaration, the type must be inferrable from
-- context.

declarations: table of declaration { full } keys (id);

declaration: record (
  id: rootid, -- id used for reference
  typename: optional_typename, -- type of object
  prag: charstring -- pragma
);

-- Each clause has a unique (within an execution environment) id and a
-- list of the statements that make up the clause

clauses: table of clause { full } keys (id);

clause: record (
  id: clauseid, -- identifier of clause
  statements: statements -- statements of clause
);

-- Each statement has a unique (within an execution environment)
-- identifier, an operator and operands, and possibly additional data
-- specified in is qualifier. Any information that is needed for the
-- proper execution of the statement and cannot be made available
-- via accessible program objects at runtime, must be provided in the
-- qualifier. The precise contents of the qualifier depends on the
-- operator.

statements: ordered table of statement { full } keys (id);
statement: record (  
id: statementid,  
operator: operator,  
operands: objectnames,  
qualifier: qualifier,  
prag: charstring  
);

-- Here are all the Hermes operators

operator: ordered enumeration (  
'add', 'and', 'assert', 'attributename', 'block', 'call', 'case',  
'checkdefinitions', 'concatenate', 'connect', 'convert', 'copy',  
'create', 'currentprogram', 'discard', 'dissolve', 'divide', 'drop',  
'empty', 'equal', 'every', 'exists', 'exit', 'expression_block',  
'extract', 'for_enumerate', 'for_inspect', 'forall', 'greater',  
'greater_equal', 'hide', 'if', 'insert', 'insert_at',  
'inspect_polymorph', 'inspect_table', 'integer literal', 'less',  
'less_equal', 'merge', 'merge_at', 'mod', 'move', 'multiply',  
'named_literal', 'new', 'not', 'not_equal', 'or',  
'position_of_element', 'position_of_selector',  
'print',  
'procedure', 'program_literal', 'real_literal', 'receive', 'rem',  
'remove', 'return', 'return_exception', 'reveal', 'select', 'send',  
'size', 'string_literal', 'subtract', 'the_element', 'type',  
'typename', 'typestate', 'unary_minus', 'unique', 'unite', 'unwrap',  
'while', 'wrap'  
);

--- Operation Qualifiers

-- Names of all the qualifier types... comments identify which  
-- operators use each qualifier type

qualifier_types: ordered enumeration (  
'absent',  
'attributename', -- attributename literal  
'block', -- block  
'}
'constraintname', -- assert, drop
'exit', -- exit
'expression block', -- expression_block
'for enumerate', -- for enumerate
'if', -- if-then-else
'inspect polymorph', -- inspect_polymorph
'inspect table', -- for inspect, inspect_table
'literal', -- integer literal,

'program literal', -- program literal
'return exception', -- return_exception
'select', -- select
'selector', -- every, exists, extract,

'typename', -- real literal, named literal
'while', -- forall, remove,

'wrap' -- the_element, position_of_selector

); -- Following is the variant used to represent all statement qualifiers

qualifier: variant of qualifier_types(
  'absent' -> empty: empty { }, -- no qualifier
  'attributename' -> attributename: attribute_name { full },
  'block' -> block: block_qualifier { full },
  'constraintname' -> constraintname: constraintname { full },
  'exit' -> exit: exitid { full },
  'expression block' -> expression: expression_qualifier { full },
  'for enumerate' -> for enumerate: for enumerate qualifier { full },
  'if' -> if: if qualifier { full },
  'inspect polymorph' -> inspect polymorph: inspect polymorph qualifier { full },
  'inspect table' -> inspect table: inspect table qualifier { full },
  'literal' -> literal: charstring { full },
  'program literal' -> program literal: processid { full },
  'return exception' -> exceptionid: exceptionid { full },
  'select' -> select: select qualifier { full },
  'selector' -> selector: selector { full },
  'typename' -> typename: typename { full },
  'while' -> while: while qualifier { full },
  'wrap' -> formal typestate: formal typestate { full }
);
A block statement qualifier identifies the scope holding the declarations and main body code of the block, the root objects held constant within the block, and all the handlers associated with the block.

```plaintext
block_qualifier: record {
    scope: scopeid, -- new scope introduced by the block
    constants: rootnames, -- objects not changed in this scope
    handlers: handlers -- exception and exit handlers
};

rootnames: table of rootname { full } keys (*);

-- Each handler names the exceptions or exit conditions that it handles, and supplies a clause (not a scope) containing the body of the handler.

handlers: table of handler { full } keys (id);

handler: record {
    id: handlername,
    clause: clauseid
};

-- Handlers come in four varieties...

handler_type: ordered enumeration (
    'builtin', -- builtin exceptions
    'user', -- user exceptions (defined with callmessages)
    'exit', -- exit conditions
    'others' -- exceptions not otherwise handled
);

-- Each handler type requires type-specific additional information to fully specify the condition handled

handlername: variant of handler_type {
    'builtin' -> builtin: builtin_exception { init },
    'user' -> user: user_exception { full },
    'exit' -> exit: exitid { init },
    'others' -> others: empty {}
Appendix C. Predefined Module

-- Here are all the builtin exceptions

builtin_exception: ordered enumeration (  
  'CaseError', 'ConstraintError', 'ConstraintFailure', 'Depletion',  
  'Disconnected', 'DivideByZero', 'DuplicateKey', 'InterfaceMismatch',  
  'NotFound', 'PolymorphMismatch', 'DefinitionError', 'RangeError',  
  'Uncopyable'
);

-- A user exception is specified by the callmessage type and one of  
-- the exceptions defined with that type

user_exception: record (  
  type: typename,  
  exceptionid: exceptionid
);

-- The qualifier for an expression block identifies the scope to be  
-- executed and the root variable that will hold the expression  
-- result. The root variable is the only variable outside the  
-- expression block scope that is not held constant within the scope.

equation_qualifier: record (  
  scope: scopeid,  
  result: rootname
);

-- For a for enumerate statement, the qualifier identifies the body  
-- scope, as well as the root variable that is used to iterate through  
-- the enumeration. The iteration variable is declared in the body  
-- scope.

forEnumerate_qualifier: record (  
  scope: scopeid,  
  enumerator: rootid
);

-- The qualifier for an if statement identifies the clause that will  
-- evaluate the test expression, the object that will hold the  
-- expression result, and the clause to be executed if the expression  
-- yields a true result. In addition, an 'else' clause may optionally  
-- be supplied, identifying the clause to be executed when the test
if qualifier: record(
  test_clause: clauseid,
  test_result: objectname,
  then_clause: clauseid,
  opt_else_clause: optional_clauseid
);

opt_else_clause: variant of option (  
  'present' -> clauseid: clauseid { init },
  'absent' -> empty: empty \ }
);

The qualifier for an inspect polymorph statement identifies the  
-- scope forming the statement body, the root variable (declared in  
-- the body scope) that will hold the unwrapped polymorph value  
-- throughout the body scope, and the typestate required for the  
-- unwrapped value.

inspect polymorph qualifier: record (  
  scope: scopeid,
  element: rootid,
  typestate: formal_typestate
);

Following qualifier is used for inspect table and for inspect  
-- statements. In addition to the same sort of selector as that used  
-- for other table operations, a body scope is included, as well as  
-- the id of the root object that holds the inspected table element  
-- throughout the body scope. The inspecting root object is contained  
-- in the body scope. Note that it IS NOT the same object as the  
-- element object in the selector itself, as they declared in disjoint  
-- scopes.

inspect table qualifier: record (  
  scope: scopeid,  
  element: rootid,  
  selector: selector
);

A select statement qualifier includes a "select clause" for each  
-- guarded arm of the select statement, identifying the guard  
-- condition for the arm and the clause to be executed when the arm
is selected. In addition, a clause to be executed when all 
guards are disabled is specified.

select_qualifier: record (  
  clauses: select_clauses,  
  otherwise_clause: clauseid  
)  

select_clauses: table of select_clause { full } 

select_clause: record (  
  clause: clauseid,  
  info: clause_info  
)  

Guard_type: enumeration ( 'boolean', 'event', 'both' );

clause_info: variant of guard_type (  
  'boolean' -> boolean: boolguard { full },  
  'event' -> portname: objectname { full },  
  'both' -> both: bothguard { full }  
)  

A boolean guard (or the boolean part of a "both" guard) identifies 
the clause to be executed to evaluate the guard, and the object 
where the result of that execution will be placed.

boolguard: record (  
  clause: clauseid,  
  result: objectname  
)  

bothguard: record (  
  boolean: boolguard,  
  portname: objectname  
)  

A selector qualifies many table-related statements, and encodes the 
information contained in the WHERE (...) clause of such statements. 
The selector introduces a scope which encodes the selector
Appendix C. Predefined Module

--- expression. The 'element' root variable appears in that scope and
--- holds a table element whenever the selector is executed. The
--- 'result' object holds the result of evaluating the selector for a
--- the table element.

selector: record (  
  scope: scopeid,       -- scope introduced by the selector
  element: rootid,     -- the element being selected
  result: objectname   -- boolean value of selector expression
 );

--- The qualifier for a while statement identifies the clause that
--- evaluates the entry test at the top of the loop, the variable that
--- will hold the result of the test, and the clause containing the
--- loop body.

while_qualifier: record (  
  test_clause: clauseid,
  result: objectname,
  repeated_clause: clauseid
 );

--- Following types definitions are for the values resulting from
--- TYPE OF and TYPESTATE OF expressions, which inspect values wrapped
--- in polymorphs

typedefinition: record (  
  typename: typename,    -- name of the type
  definitions: definitions_modules    -- resolution environment for that type
 );

typestateof_value: record (  
  typestate: formal_typestate,      -- typestate of wrapped value
  definitions: definitions_modules  -- resolution environment for
                                   -- attributes appearing in typestate
 );

Typestates and Formal Typestate

--- A typestate is a set of attributes, each of which constist of an
Appendix C. Predefined Module

-- attribute name and a list of objects that are the parameters of the
-- attribute.

typestate: table of attribute { full } keys (*);

attribute: record (    
    name: attribute_name,    
    objects: objectnames
);

-- There are three built-in attributes: init and case. An attribute
-- can also name a user-defined attribute, or constraint. The 'full'
-- attribute is an abbreviation for several init attributes, and may
-- be dropped from this list at some future time.

attribute_type: enumeration ( 'initialized', 'case', 'constraint', 'full' );

attribute_name: variant of attribute_type(    
    'initialized' -> init: empty { },
    'case' -> case: empty { },
    'constraint' -> constraint: constraintname { full },
    'full' -> full: empty { }
);

-- A formal typestate is like a typestate, but no root variable is
-- mentioned. Instead, the component lists are interpreted relative
-- to some variable (not necessarily a root object) depending on
-- context. An empty component list indicates the associated variable
-- itself, rather than any of its (sub-)components.

formal_typestate: table of formal_attribute { full } keys (*);

formal_attribute: record (    
    attribute_name: attribute_name,
    parameters: formal_object_list
);

formal_object_list: ordered table of component_list { full };

end definitions
C.1 References


Appendix C. Predefined Module


Index

access control, 53
accuracy requirement, 103
and, 105
approximate arithmetic, 103
attribute, 132
    name resolution, 78
attribute arguments, 84, 132
attribute compatibility rules, 86
attribute not present, 92
base variable
    multiple declarations of, 76
block, 10
block, 96–97
boolean
    guard, 26, 98
    literals, 50
    type family, 102
builtin, 75
    attribute names, 84
    exception names, 78
    typestate attributes, 78
call
    arguments, 22
    forwarding, 60
    interface, 22
    parameters, 22
call, 5, 9, 118, 121
    arguments and parameters, 119
    exception names, 78, 119
    returning typestate, 122
callmessage, 8, 22, 94, 118
    constant components, 88, 119
    discarded, 29
discarding, 95
    init, 87
type, 24
callmessage type
    definition, 119
cannot add, 92
cannot coerce, 92
cannot drop, 92
capability, see output port
capability-based
    case, 61
case, 78, 84, 86
case of, 117
case typestate, 86
casetypestate attribute, 62
CaseError exception, 63
Charstring, 19, 28
checkdefinitions, 131
checked, 78, 84, 124
checked program, 23
checkeddefinitions, 78, 84, 131
class, 8
class rule, 151
class rules, 55, 80, 81
child, 9
component name resolution of, 77
components, 107
concatenate, 109
connect, 14, 121
copy, 88
constant, 22
constant copy, 88, 112, 113, 119, 127
costant, 131
ConstraintError exception, 132
ConstraintFailure exception, 132
conversion and resolution, 72
convert operator, 106
copy of, 29
copying a value, 94
create, 124
create of, 15, 16, 24
currentprogram, 129
declaration, 18, 19
        hiding not allowed, 20
declarations list, 76
definition, 18
definitions module, 21, 77
definitions modules, 19
Depletion exception, 95
discard, 57, 95, 95
Discarded exception, 29, 119, 123
discarded exceptions, 78
Disconnected exception, 29, 99, 122, 123
dissolve, 116
DivideByZero exception, 104
DuplicateKey exception, 43, 48, 111, 112
element, 107
element type, 107
else, 97
empty, 61, 120
enabled clause, 99
entry typestate, 118
enumeration type family, 102
evaluate, see expression block
event guard, 26, 98
every, 110
every of, 42
exception, 11, 119, 122
        definition, 22
        handler, 11, 96
        name resolution, 78
        return, 123
exists, 110
exists of, 48
exit
        name resolution, 79
exit, 79, 88, 123
exit, 101
exit formal typestate, 119
exit typestate, 24
expression block, 59, 88, 100–101
extract, 42, 111
fairness
        in receive statement, 122
        in select statement, 99
filter program, 12
for enumerate, 106
for statement, 46
for-inspect, 113
forall, 110
formal typestate, 86
full, 23, 78, 85
function
        call, 121
        function call, 122
        function notation, 12
generator
        process, 125
generator process, 125
guard, 98
guarded select, 26
handler clause, 79, 96
Hermes, 3–5
research directions, 67
type system, 55
hide, 117
hide statement, 68
Hoare, C.A.R., 55

identifiers, 74
if, 97
imports list, 19, 78
inference function
  elementtypeof, 151
  sameas, 152
inference rule, 151
inference rules, 21, 80
inferred declaration, 81
init, 23, 78, 84–86
init(), 56
initialization port, 8, 19
import, 118
input port, 8, 22, 94, 118, 119
  as event guard, 98
  definition, 23, 24
  enabled, 26
insert, 29, 111, 111
insert-at, 111
inspect, 46, 112
inspect variable, 112
integer literals, 50
integer type family, 103
interface, 11, 22, 60
  definition, 22, 24
InterfaceMismatchException, 17, 124, 124

key, 108, 110–112
keys(), 48
keyword, 74

late binding, 5
lexical analysis, 71
linking list, 51, 79

literals, 104
long-lived systems, 55

main clause, 96
main program, 124
meet, 89
merge, 112
merge-at, 112
message type, 118
minimum formal typestate, 119
  default, 119
minimum typestate, 87
mod, 104
module, 4
move, 99
moving a value, 93
moving versus copying, 29, 93

name equivalence of type definitions, 24
name space, 75, 78
  of base variables, 76
  of component names, 77
  of type identifiers, 77
  of user-defined exception names, 78

named literal, 73
new, 14, 107, 119, 120
nominal type family, 101, 105
normal exit, 96
not, 105

notation
  association, 121
  function, 121
  positional, 121
NotFound exception, 42, 110–112
numeric literal, 73

object-oriented languages, 27

occurrence
  applied, 72
Appendix C. Predefined Module  
217

predefined, 72
on (others), 50
on (others), 96
operating system, 4
or, 105
ordered

table operations, 29
ordered, 108
ordered table, 28
ordered table, 28
others, 96
otherwise, 27
outcome, 88
output, 119
output port, 8, 22, 119
definition, 24
overlap, 121

parent, 8
passing parameters by value-result, 12
pointers, prohibited, 4
polymorph, 94
polymorph, 64
polymorph type family, 125
polymorphic types, none, 18
PolymorphMismatchException, 66, 126, 127
position, 41
position in a table, 28, 110–112
position-of-element, 92
position-of-element, 113, 114
position-of-selector, 114
position-of-selector, 42
postcondition, 89
postcondition function, 90
precondition, 88
predefined, 19, 75
character type, 102
predefined, 19, 78
predefined exceptions

DivideByZero, 104
predefined! boolean, 97
predefined! program, 51
procedure, 40

procedure, 125
procedure of, 40
procedures, 40
process, 4
communication, 12–17
module, 8
name, 79
termination, 29
process, 19
block, 96
program, 15
checked, 23
program, 15
program literal, 50
program value, 71, 80
punctuation, 73, 74

qualifier, 80, 81

RangeError exception, 29, 106, 111, 112

real

literal, 73
type family, 103
real arithmetic, 103
receive, 5, 9, 129
record, 107
record definition, 43
rem, 104
remove, 29, 42, 111
repeat clause, 97
reserved words, 74
resolution, 75

of base variables, 76
of component names, 77
of type names, 77
result variable, 100
return, 5, 9, 24, 123
exception, 78
reveal, 63, 117

safe number, 103
safe numbers, 103
scalar type families, 101
scope, 76, 80
of element variable, 109
of inspect variable, 112
security, 55
select
no fair choice, 99
select, 26
abbreviation, 40
select, 98–100
selector, 41, 91, 109
expression, 109
long, 109
mapping, 109
semilattice, 57, 84
meet, 57
send, 118, 124
send statement, 60
server, 9
server process, 25
shared data, prohibited, 4
shell, 9
size of, 26, 114
space, 73
string literal, 50, 73, 109
syntax
analysis, 71
systems programming, 12
table, 28
new, 109
comparison, 108
element typestate, 108
insert, 29
insert at, 29
operations, 28
representation independent, 28

type family, 107
unordered, 43
table[expr], 42
table[key], 43
the-element, 42, 110
tokens, 73
type
checking, 8, 9, 17, 18, 72, 80–82, 150–153

definition, 22, 77
families, 8
inference, 72, 80–82
inference rules, 55
inferencing, 150–153
name, 19
specifier, 21
type, 127

type name, 77
resolution, 77
type specifier, 81, 82, 83
o, 104
typestate, 82
case attribute, 62
attributes, 23, 56, 78, 84–85
checking, 9, 17, 24, 56, 72, 84–92, 150–153
checking algorithm, 90
checking example, 89
coercions, 87
constraint attributes, 85
entry, 23, 24
errors, 91–92
exit, 23
exit formal, 119
formal, 85–86
initialization, 23
minimum formal, 119
postcondition, 90
postcondition rules, 152
precondition function, 89
precondition rules, 152, 155
semilattice, 84
semilattice meet, 87
syntax, 84–85
valid, 86
typestate, 127

Uncopyable exception, 64, 94
unique, 105
unite, 116
unite statement, 62
unwrap, 66, 126
unwrap statement, 84
using list, 19
variable
  discarded, 29
  finalized, 29
  name resolution, 75–77
  overlap, 121
  temporary, 122
variable name
  base variable, 75
  component name, 75
  formal, 86
variant, 61
  hidden, 61
  revealed, 61
  type family, 115
  typestate, 86
visibility, 75

where, 109
where expression, 41
where('true'), 47
while, 10
while, 97–98
window system, 31
word, 73, 74
wrap, 66, 125
wrapped value, 125
wrapper, 125