- Signed integer and real literals are incorrectly parsed. A plus or minus sign (+ or -) immediately preceding an integer literal is tokenized as part of the literal in all cases. This means that expressions like "a+1" end up getting parsed incorrectly (the parser sees a variable followed by an integer literal with no intervening operator). The workaround for this bug is to interpose a space between the sign and the first digit of the literal (as in "a + 1").

- Storage leak in stdio module. Every call to the stdio C-Hermes process allocates a selector frame on the process' stack frame, which never gets released. This is most noticeable when the BLOCKINGIO environment variable is not set, since then all IO requests will be handled by polling for data. Thus, for example, an idle shell process could end up consuming all available memory and eventually die with a malloc failure. Setting BLOCKINGIO keeps the rate of leakage low enough that it should not normally be a practical problem.

- Inconsistent and/or unhelpful error messages. Some of the Hermes system modules (like root, cache, shell, etc.) either do not check for common errors or do so inadequately (like via an "on (others)" clause at the end of the module), and therefore produce terribly inadequate diagnostics when errors arise.

- Carriage return not a line terminator. A carriage return should be recognized as an end-of-line character by the compiler. The line counter should be incremented for each CR not appearing with an accompanying LF, and for each LF.

- Incorrect reports of dead code. If a process module contains only statements that cannot raise exceptions, the compiler will complain about dead code in the block and refuse to compile the program. This is due to an implicit "on(others)" handler that is wrapped around the entire process body. If the process is immune from exceptions, that handler cannot ever be entered. Note that explicitly coding an "on(others)" handler on the process outer block will not trip this bug, because entering that block can itself raise Depletion.

- BLOCKINGIO mandatory. The current implementation of nonblocking IO appears to be extremely buggy. Symptoms range from the parent Unix shell exiting when Hermes terminates, to segmentation faults when the compiler is unable to locate source files. Users should always have the BLOCKINGIO environment variable set (to anything) when running Hermes until further notice.

- Expression block scope violation. The result variable appearing in an expression block (evaluate...from statement) is currently placed in the scope containing the expression block, rather than the new scope created for the expression block. This means that you cannot currently have two expression blocks naming the same result variable in overlapping scopes.

- **Position of** again. This has the same underlying cause as the previous bug. If you inspect an element of an ordered table, delete the element you're inspecting, and then attempt to find its position via **position of**, you will probably get a result that is one greater than the size (after the remove) of the table. You may get a different answer if there is another table element equal to the one you were inspecting.

- Inspected polymorphs not modifiable. In INSPECT-POLYMORPH, the unwrapped object should be held constant, but the polymorph itself should not. Currently, modifying the polymorph in the body of an INSPECT-POLYMORPH is not supported, and doing so will likely cause the interpreter to crash.

- Depletion handling. If memory runs out during execution, a **depletion** exception should be raised. This has not been adequately tested, so may be unreliable.

- Failures cause Hermes to abort. Many failures that may occur during the operation of the Hermes interpreter are not yet properly handled. In many cases, the current implementation simply shuts down, even when a reasonable alternative exists. All such situations are considered to be extremely unlikely, and should be fixed in due time.

- Deadlocks in distributed applications. If two interpreters send each other very large messages, they may deadlock due to flow-control mechanisms in the underlying communications protocols. This is because synchronous `write` operations are used to send messages.

- Inports incorrectly garbage collected. Updating of reference counts for inports with remote connected outports is subject to a race condition that can cause the interpreter to crash. This has never been observed in practice.

- File descriptor depletion. Programs that create many windows or link to several remote interpreters may cause the interpreter to run out of file descriptors. There may be cases where this will cause the interpreter to either hang or crash.

- Distributed interpreters never exit. An interpreter in which the `start network` C-Hermes process has been invoked will never terminate, waiting instead for incoming messages from the network. Typing control-C at the controlling terminal should halt such an interpreter.

- No user exceptions from C-Hermes processes. Our mechanism supporting Hermes system processes written in C does not allow the return of user-defined exceptions. Therefore, such processes are incapable of returning the exceptions defined with the interfaces to which they otherwise adhere. In all such cases, the processes currently discard the callmessage.

- Nominals. Two invocations of the interpreter may generate identical sequences of nominals, though this is highly unlikely. The invocations must sample the **UNIX** system clock within the same millisecond to achieve this effect. If they sample the clock within different seconds, their sequences are guaranteed disjoint for at least the first 3 billion nominals. If they sample the clock within the same second, but N milliseconds apart, the earlier invocation must generate N million nominals before its sequence overlaps the sequence of the later invocation. An invocation samples the clock initially when generating its first nominal. It samples it again after 3-4 billion nominals, and thereafter in a period of 2^{32} nominals.

- Function calls are expressions and should only be able to modify their result argument; however, the abstract program fails to distinguish between function calls and call statements, rendering this check impossible.

- Reals. The accuracy attribute is not supported. An out-of-range real literal is compiled as real zero, without diagnostic.

- Checkdefinitions. The **checkdefinitions** operation is not supported, nor is the **checkeddefinitions** typestate attribute.

- Coercion of Callmessages and Variants. At typestate merge points (such as the end of an conditional statement or the beginning of a loop) coercions may be inserted by the typestate checker to discard callmessages and hide variants. These operations are incompletely implemented and may lead to values not being discarded when they should. This is only a problem for inport, outport, and callmessage values, as these are the only types for which discards can have visible side-effects.

- Copying tables containing uncopyable objects. If a table that might contain uncopyable objects (inports or callmessages) is copied, it should generate a run-time **Uncopyable** exception; however, the current implementation will not raise the exception until either the original or the copy is modified (due to copy-on-write sharing of tables), meaning that the exception might be raised at the wrong place (such as an **insert**).

- Hidden table copying. Some Hermes operations that are not allowed to raise **Uncopyable** exception nevertheless may, because their current implementation involves copying potentially uncopyable objects. This includes **inspect_table**, **for_inspect**, and **extract**.

- **Position of** bug. This is a very arcane bug: if you inspect an object in a table using positional notation, and then use the **position of** operation on it, if there are other objects in the table with the same value, you may get the position of one of the equal values instead of the one you actually have.

```
hermes cache \; startnet 3 \; linkserver \; postprinter
```

2. On a machine called "blue," run the `print` program after first chaining onto red:3's resource manager:

```
hermes cache \; startnet \; getlink red:3 \; print file.txt
```

Note that these facilities do not permit the construction of circular resource manager chains; each `getLink` operation can only chain to resource managers that already exist. It would be possible to program arbitrary configurations using the C-Hermes functions directly.

8 Unsupported Features and Known Bugs

This section summarizes the Hermes language features which are not yet supported in the current implementation, as well as bugs known to exist.

- Constraints. There is no support for user-defined constraints and the **assert** and **drop** statements.
- Checking of Polymorphs. Typestate information is not associated with a polymorph value, so polymorphs are not completely checked and hence insecure. This means that the **typestate of** operator does not work properly, and that unwrap and inspect-polymorph do not raise the mismatch exception for incompatible typestates.
- Constant lists on blocks are not supported.
- Parameter lists using association (also called keyword) notation are not supported; only positional notation may currently be used.
- Programs as objects. Program objects that are written by the compiler (the `*.po` files) do not include the abstract program object; thus all the fields of objects of type **predefined!program** are not initialized. Accessing these fields will cause unpredictable results.

  In addition, the **checked** attribute is not supported because it was intended to be a constraint.

7.3 The `LinkServer` Program

`LinkServer` is a `sysMain` program that allocates a port for receiving linking calls from other interpreters, and performs the `"set linking port"` C-Hermes call to identify this port to the interpreter. It then spawns a process which accepts all future requests made to this port. Each request is returned carrying a copy of the (unfiltered) resource manager that was provided to `linkServer` via the `sysMain` interface. The remote process that made the linking request is thus able to retrieve any capabilities or other resources that were posted to that resource manager.

7.4 The `GetLink` Program

`GetLink` is a `sysMain` program that takes any number of command line arguments. Each argument should be of the form "*hostname*:*interp-no*, where *hostname* and *interp-no* identify a remote Hermes interpreter. If the colon and *interp-no* are omitted, the interpreter number defaults to 1.

`GetLink` issues a `"get link"` C-Hermes call in order to establish interoperability with each of the identified remote interpreters. It calls each linking port and, assuming the remote interpreters have run the `linkServer` program, each remote interpreter returns copy of its own local resource manager. `GetLink` chains all these resource managers, together with the one it received via its own initialization call on the `sysMain` interface, and replaces the resource manager from its initialization call with the chained resource manager. Subsequent programs invoked from the same command line by `root` will thus be able to retrieve capabilities posted to the remote resource managers to which it has linked.

7.4.1 An Example

Here is the printer example that was described in Section 5.2.6, recast in terms of the programs shown in this section. Suppose that `postPrinter` is a program that simply posts a capability to a print spooler to what it believes is its local resource manager. And suppose that `print` is a program that retrieves a print spooler resource from what it believes is its own local resource manager and then uses it to print a file. We can arrange for these two programs to run in separate machines and still work without modification as follows:

1. On a machine called "red," run the `postPrinter` program after running `startNet` and `linkServer` so other interpreters can interoperate:

by trial and error: after loading the named program, it first tries to instantiate it as a `main` process. If that results in an `interfaceMismatch` exception, it tries again, this time assuming `sysMain`. If that fails, `root` gives up and complains.

The `cache` is really a `sysMain` program, not a `main` program. It operates by chaining the resource manager it gets on its initialization call with a new resource manager that it creates. The cached versions of `pathLoad`, etc. are posted to the new resource manager, and it is the chained resource manager that is passed on to the user program. `Cache`, like `root`, is capable of running both `main` and `sysMain` programs. When it runs `main` programs, it uses the same code as `root` to decide on the user name for the filter, based on Unix environment variables.

Now here comes the tricky part: the `root` process is capable of running several commands on the same command line. The individual commands must be separated by semicolons, and the semicolons must be separate "words" in the command line, i.e. they must be surrounded by whitespace.[13] The commands are run in sequence, and a `sysMain` program is actually allowed to alter the resource manager that will be provided to following commands. This is achieved by making the `rm` component of the `sysMain!sysMain` callmessage be non-constant on the interface. A `sysMain` program is thus free to alter this component of the callmessage before returning it.

The cache makes use of this facility. Before returning its callmessage, it replaces the incoming resource manager with the chained resource manager that it created. Thus the commands `hermes cache foo`, and `hermes cache \; foo` should behave identically.

Note that the shell is a `main` program and is only capable of running `main` programs. The shell does not treat semicolons on its command lines as command separators.

Now we can continue the discussion of the interoperability features.

7.2 The `StartNet` Program

The `startNet` program performs the `"start network"` C-Hermes function, initializing networking support and supplying an interpreter number for the current interpreter. `StartNet` expects a single command line argument, which should be an integer. This is the number it supplies as an interpreter number. If the command line argument is omitted, the default of 1 is used.

[13] Be careful to quote the semicolons, e.g. with backslashes, to prevent their being interpretered by your Unix shell.

The following code fragment creates a window and uses it to retrieve the user's name, then destroys the window.

```
linking (safeWindow)
...
unwrap CLoad from args.rm.get("CLoader", "") {init};
openWindow <- (create of process safeWindow)(CLoad);
window <- openWindow("Query", "-Wh 1 -Ww 30");
call window.terminal.terminal.putString("Your Name: ");
name <- window.terminal.terminal.getString();
discard window;
...
```

7 Distributed Hermes

Section 5.2.6 described three C-Hermes functions that could be used to provide interoperability among several Hermes interpreters. In this section we describe three corresponding Hermes programs that make use of these facilities to provide an easy, flexible mechanism for establishing interoperability. The programs are called `startnet`, `getlink`, and `linkserver`. Sources for all can be found in the `tools/distrib` directory.

Before describing these facilities it is necessary to come clean on some information that was held back earlier in this User's Guide.

7.1 The Truth About the `Main` Interface, the `Root` Process, and the `Cache`

We said earlier that the `root` process (the first process that's spawned by the interpreter at startup) is responsible for running whatever program is named on the command line, and that this program must obey the `main` interface. This is not exactly correct. In truth, the `root` process will run programs written to either of two different interfaces. `Main` is one of them; the other is `sysMain`. `Main` and `sysMain` differ primarily in that the `main` interface provides a filtered resource manager, while `sysMain` provides an unfiltered one. The `sysMain` interface is thus suitable for programs like login shells that need to be able to construct filtered resource managers. The `root` process knows which interface to use

achieved without requiring P to block.

6.2.4 Bugs

Be cautious about file system links interacting with `MKSCOPE` and `HPATH`. If links cause a file to appear in one directory according to `HPATH` and another directory according to `MKSCOPE`, then makehermes will never recompile the file.

Hermes supports two kinds of comments: `--` and `/* */`. If the latter type of comment appears in a source file before the `using` list or before the `linking` list, makehermes will behave unreliably.

6.3 Windows

If you are running Hermes on a SUN workstation with Sunview or an IBM RS/6000 with X Windows, you may create *tty windows* from your Hermes program. You may then use the `terminalio` operations `PutString`, `PutLine`, `GetString`, `PutChar`, and `GetChar` to read from and write to these windows.

The preferred method to open a window is via the `safewindow` program that is supplied with the system. After creating a process running this program, make a call on its initialization port (of type `safeWindowInit!safeWindowInitFn`), passing the `"Window"` C-Hermes capability. In return, you will recieve a capability of type `safeWindow!safeWindowFn` that can be used to create many windows.

To create a window, call the capability you received from `safewindow`, passing it a label string for the window's title strip and a string specifying window options (size, colors, position, etc.) in the style used by SunView.[11] This call will return a callmessage containing a `terminalIO!terminalFunctions` record containing capabilities to print to and read from the window. You can copy and give away these capabilities as you see fit so that other processes can perform I/O to your window. When you wish to destroy the window, simply discard the callmessage you got from the window creation call.[12]

[11] See the section named "Generic Tool Arguments" in the sunview(1) man page.

[12] This is an example of a fairly tricky method by which a process P can send capabilities to another process Q and be guaranteed to receive notification when Q is finished with the capabilities. P creates a "proxy" process and gives it capabilities to two of P's ports, A and B, as well as whatever data P intends to send to Q. The proxy process calls P on its port A, passing it the data intended for Q. P then sends this data, wrapped in the callmessage received from the proxy, to Q in whatever fashion is appropriate. When Q is finished using the data, it discards the callmessage. This causes an exception to be raised in the proxy process, and it can then call P back on port B to notify P that Q has finished. This scheme has the advantage that Q cannot terminate and "forget" to notify P; if Q terminates, the callmessage it received from P will be discarded automatically, and the notification will take place as required. This is

These options can be combined, as in:

- `hermes makehermes -agp bar.po` will print out all the dependencies for `bar.po` and ignore errors.
- `hermes makehermes -og foo.do` will compile all inconsistent files used by `foo.do`. If a compilation error occurs when compiling a dependency `guppy.d`, then files dependent on `guppy.d` will not be compiled but other files are compiled.

6.2.3 Delimiting the scope of makes

It is sometimes the case that one does not want makehermes to recompile all inconsistent files. This may be due to several reasons: for instance, you may be importing your friend's definitions modules, and she is in the process of updating source files. Any attempt to compile those source files would fail, because they are in a changing and inconsistent state. Instead, you wish to simply use the most recent compiled versions of the modules and prevent `makehermes` from attempting to update them.

`Makehermes` uses a list of directories to limit the set of files it compiles. It compiles `only` those files stored in directories in the list. The directory list is specified as a standard unix path specification in the environment variable `MKSCOPE`. Any file *not stored in* a directory listed in `MKSCOPE` is considered consistent regardless of its modification time. That is, makehermes pretends that the modification time for a file not within `MKSCOPE` is infinity.

For example, suppose that `MKSCOPE` is set to .:/hermes/lib:/u/pip, and that process module `bar` imports the definitions module `foo`. Assume further that `bar.po`'s full pathname is `/u/pip/bar.po` and that the module `foo.do` is in a directory not listed in `MKSCOPE`. Then makehermes will compare the timestamp of `bar.po` with the timestamp of `foo.do` and recompile `bar` if necessary. However, it will not recompile `foo`, even if `foo.do` has a timestamp that is earlier than its source file or any module on which it depends.

If not defined, MKSCOPE defaults to the current working directory, so that `makehermes` will not normally attempt to recompile anything that does not appear in the current directory.

everything is finished, and no module needs to be compiled twice. Inconsistencies are determined by examining the modification times maintained for the source and object files by Unix.

Makehermes searches for files (both compiled modules and source files) in the directories listed in the Unix environment variable `HPATH`. If `HPATH` is not defined, makehermes searches on the path `.:`*hermes-root*`/bin`, where *hermes-root* is where the Hermes system has been installed. The object files produced by compilations are always placed in the current working directory, regardless of where the source files may have been found.

If a required object file (`.do` or `.po` file) is missing, `makehermes` will consider that module inconsistent and will therefore recompile it.

6.2.2 Options

Makehermes recognizes three categories of options. Options are specified on the command line by means of one or more option letters preceded by a hyphen. The options are as follows; defaults are marked with asterisks:

1. The *action* options determine what makehermes will do with an inconsistent file.

```
*c      (compile)          compile (pcom or dcom) the file
 p      (print)            print the filename and compilation
                           command, but do not compile
```

2. The *error* options control makehermes's response to errors encountered while building the dependency graph and applying the required actions.

```
*s      (Stop)             halt immediately on any error
 g      (Ignore)           ignore all errors
 o      (Continue Others)  on an error stop processing successors but
                           continue processing other paths
```

3. The *selection* options help select the files to which makehermes will apply its actions.

```
*i      (Inconsistent)     apply the action to all inconsistent files
 n      (None)             take no action (just build the dependency graph)
 a      (All)              do action to all files (in order)
```

The interfaces for the `personalRM`, `chainedRM`, and `chainedSysRM` programs can be found in the `tools/rmanager` directory instead of the `interface` directory; `personalRM` interfaces are in `personalrm.d`, while `chainedRM` and `chainedSysRM` interfaces are in `chainedrm.d`.

6.2 Make facility

The `Makehermes` program helps automate the compilation of a set of interdependent modules. Makehermes is easier to use than the unix make program because it automatically determines module dependencies from the `using( )` and `linking( )` lists in Hermes modules. The user need not create a "makefile."

makehermes is invoked by the command:

`makehermes` *[options] filename ...*

Makehermes determines and performs (depending on its options) the necessary compilations (both `dcom` and `pcom`) to produce the requested compiled module files. Process modules and definitions modules may both be specified, with the appropriate file extentions (`.do` for definitions modules, `.po` for process modules). For example, the following command would compile one definition module and three process modules, as well as any other modules on which they depend and which are out-of-date.

```
makehermes foo.do goo.po /u/hermes/bar.po zip.po
```

6.2.1 Module dependencies and consistency

A module `bar` *depends on* another module `foo` if either `foo` is a definitions module imported by `bar` (in its `using( )` list), or `foo` is a process module statically linked by `bar` (in its `linking( )` list). `bar` is *inconsistent* if its source file was modified since it was last compiled, or if it depends on any other module that is inconsistent or was compiled more recently than `bar`.

The `makehermes` program determines all the modules on which each of the `filename` arguments depends, and then all the modules on which those modules depend, and so on. In this way it creates a *dependency graph* where nodes represent modules and edges represent dependencies between modules. `makehermes` then recompiles all the inconsistent files represented in the graph, in an order that ensures that all files will be consistent when

Given an unfiltered resource manager, to build a system in which no client of the resource manager can cheat, it's easy to create filters for given users with the `userRM` program:

```
...
filteredRM <- userRMFn#(create of userRM)("Joe", bareRM);
...
```

You can then pass `filteredRM` to any of Joe's processes, confident that they will not be able to break the security of your access control functions.

There is a convenience program called `personalRM` that creates a new resource manager along with a filter for a specified user. It performs the above two operations in a single step:

```
...
filteredRM <- personalRMFn#(create of personalRM)("Joe");
...
```

Finally, there are facilities for "chaining together" multiple resource managers. Chaining can be done for both filtered and unfiltered resource managers.

A chained resource manager is a process that maintains an ordered table of resource managers of the same type. Whenever a retrieval request (`get` or `remove`) arrives, the chained resource manager offers it to each of the resource managers in its list in turn, until one of the resources responds with either success or an `accessDenied` exception. All other requests (`post`, `insert`, and `delete`) are sent only to a single resource manager in the list, as designated when the chained resource manager was created.

To chain resource managers, collect the component resource managers together and then instantiate and call the `chainedRM` (or `chainedSysRM` for unfiltered resource managers) process module:

```
...
new RMList;
insert copy of RM_1 into RMList; -- this is the RM for post requests
insert RM_2 into RMList;
...
chainedRM <- chainedRMInitFn#(create of chainedRM)(RMList, RM_1);
...
```

6.1.2 What About User Names

User names have been mentioned a few times in discussing the resource manager, but they have not appeared on any of the resource manager interfaces. This is because we have really been discussing the interfaces seen by "filtered" access to the resource manager, rather than direct access. The resource manager really exports services as defined in the `interface/sysRManager.d` definitions module. Each interface (`post`, `get`, etc.) in the `sysRManager` definitions module includes a `userName` argument.

Normally, when a process is given access to a resource manager, it is not given direct access. Instead, a "filter" process is created that will serve as an intermediary for all calls to the resource manager. The filter is supplied with the direct resource manager capabilities and with the name of the user whose requests it will filter, on its initialization call. That call returns a set of filtered resource manager ports (corresonding to the `rManager`, not the `sysRManager` interfaces), and it is these ports that are provided to the application process. From that point on, the filter process will translate every incoming call on the filtered interfaces into an unfiltered call directly to the resource manager, and it will add the user name it got in its initialization call to each such request. Because Hermes is a safe language and capabilities are unforgeable, the application program cannot circumvent this arrangement and pose as another user. Thus the access control functions are able to trust the identification information accompanying retrieval requests and make decisions based on it.

6.1.3 Creating Resource Managers

To create a resource manager and get direct, unfiltered access to its services, instantiate and call the `rmanager` process module:

```
declare
  RMInit: sysRManagerInitFn;
  RM: sysRManager;
  ...
begin
  ...
  RMInit <- create of process rManager;
  RM <- RMInit();
  ...
```

To retrieve a resource, call the `get` service of the resource manager, passing the name of the desired resource and whatever parameter string is required by the access control function guarding the resource (the `ac_none` access function ignores the parameter string, so you can just pass an empty string for resources guarded by `ac_none`):

```
...
unwrap jokeOfTheDay from args.rm.get("joke of the day", "") {init};
call jokeOfTheDay(...);
...
```

To delete a resource that you posted, call the `delete` service of the resource manager with the name of the resource to delete:

```
...
call args.rm.delete("joke of the day");
...
```

The `insert` and `delete` services of the resource manager are artifacts of the ugly blemish of uncopyable objects in Hermes. The problem is that the normal resource manager services require that resources be copied in certain circumstances. For example, the `post` service interface indicates that the `resource` argument is full and is initialized on both the call and the return. This makes it necessary for the resource manager to copy the resource in order to save it in its repository. In addition, the resource manager always passes a copy of the requested resource to the associated access control function when a `get` request is made, so that the resource can remain in the repository for future requests. If there were no other services provided by the resource manager, this would imply that uncopyable objects could not be posted as resources to the repository.

Consequently, the `insert` and `remove` services correspond to the normal `post` and `get` services, but they do not attempt to copy the resources. This means that if you post a resource by means of the `insert` service, the resource will be unititialized for you following the call. It also means that only a single retrieval request can be made by the `remove` service, since the resource cannot be given away and still remain in the repository for future requests.

retrieval request; if there is no such resource, an exception return is made to the requestor (`rManager!getResource.notFound`).

2. The resource manager calls the access function associated with the resource, passing it: (1) the name of the requesting user; (2) a copy of the resource from the repository; and (3) the parameter string supplied in the retrieval request.

3. The access function decides whether or not the requestor should be allowed access to the resource, and if so, returns a resource (a polymorph) to the resource manager.

4. The resource manager passes the resource returned by the access function back to the requestor.

Note that the resource returned by the access function in step 3 need not be the resource that was stored in the repository. The access function is free to return whatever it wishes as the resource that will fill the retrieval request. A customized resource can be built "on-the-fly," or access to a capability can be provided through a "filter" process so that the access can be revoked if needed at a later time.

If, in step 3, the access function decides that the requestor should not have access to the requested resource, it can make an exception return (`rManager!access.accessDenied`), which then turns into an exception return (`rManager!getResource.accessDenied`) to the requestor.

6.1.1 Using the Resource Manager

A resource manager normally comes in the form of a record of type `rManager!rManager`, which contains five component capabilities: `post`, `get`, `delete`, `insert`, and `remove`. The interfaces for these interfaces are all defined in `interface/rmanager.d`.

If you wish to post a resource, choose or write an access control function.[10] Wrap the resource into a polymorph (it usually is an output port but need not be), and call the `post` service of the resource manager:

```
...
ac <- create of process ac_none;
wrap copy of jokeOfTheDayFn as resource;
call args.rm.post("joke of the day", resource, ac);
...
```

[10] We supply a "trivial" access function `ac_none`, which will give anybody who asks a copy of the resource.

of access rights may be very large, and in some cases it may be open-ended—that is, new services may be added and you may wish to give the child access to them without extending the parent-child interface and recompiling both parent and child. The Hermes shell is a good example of such an application. Since it is intended to run arbitrary programs, it has no way of knowing which services will be required by any given program. Trying to provide all known services via the initialization port is unwieldy and difficult to extend.

We address this problem by supplying a program called the resource manager. A process running this program owns a repository holding a dynamically varying collection of resources (usually capabilities to make calls to various services), each identified by a name in the form of a character string. The resource manager provides services that allow other processes to add and remove resources, and to retrieve resources by name. Instead of passing a newly created process a fixed set of capabilities to services, the creating process can pass capabilities to a resource manager. The new process can later request particular resources from the resource manager. The requested resources will be delivered if they are in the repository at the time of the request and if the resource manager determines that the caller has authority to obtain the ports. Our approach is based on the resource repository example in the Hermes Tutorial.

Since resources will in general have different types, they are passed always wrapped in polymorphs (of type `common!polymorph`) when being passed between applications and the resource manager. An application wishing to add a resource to the repository must `wrap` its value before calling the resource manager's `Post` service. Likewise, a process that retrieves a resource from the repository will need to `unwrap` it before it can be used.

The resource manager maintains three additional pieces of information for each resource that it manages:

- the resource name;
- the "owner"—the name of the individual who posted the resource; and
- an access control function.

When a resource is posted to the repository, all three pieces of data, along with the resource itself, are supplied as call arguments to the `post` service. When a retrieve operation is performed, two arguments are supplied: a resource name, and a "parameter" string. The retrieval request is handled as follows:

1. The resource manager attempts to locate a resource with the name specified in the

This function accepts a `printf`-style format specification and a real number, and produces a character string displaying the number. The function is written as a C-Hermes function rather than as a Hermes process primarily for expediency. It's a somewhat complicated function to write, and it's already available via `sprintf` for C programs. As an example, to print out a real number, left-justified following a comment, with at most two digits folowing the decimal point:

```
...
unwrap rtoa from CLoad("crtoa") {init};
call putLine("The value is: " | rtoa("-10.2g", x));
...
```

5.3 Unix Stdio Functions

This section describes the capabilities available in the `stdio` component of the standard "`unix`" resource.

This section is not yet written. For a list of the capabilities, see the `unix` *definitions module in* `interface/unix.d`*. Most of the capabilities have names corresponding fairly closely with standard Unix system calls and library routines. The actual implementations are in a combination of the C-Hermes process code (*`cfunc/stdio.ch`*), and a Hermes process wrapper that can be found in* `hermes/io/stdio.pp`*. Good luck!*

6 Tools

Included in the distribution of the Hermes system is a set of tools for building applications. Some of these tools can be found in the directory hermes/tools. We will describe four tools below: a resource manager, a Hermes make facility, a windows package, and a capability for communicating with Hermes processes executing on remote machines.

6.1 Resource manager

When you initialize a Hermes process by calling its initialization port, you often pass some output ports to give the new process initial access rights. In some applications the set

```
begin
  ...
  unwrap startNet from CLoad("start network") {init};
  call startNet(interpNo);
  unwrap getLink from CLoad("start network") {init};
  link <- getLink(remoteHost, remoteInterpNo);
  wrap empty as request;
  call link(request);
  unwrap printer from request {init};
  call printer(file);
  ...
```

This example is not terribly attractive because of the restriction that only one linking port can be established for each interpreter. In this case, because the linking port is being used to provide access to a printer, it cannot be used for exchanging any other services. The higher-level functions described in Section 7 make use of resource managers to provide a more general facility.

5.2.7 Parsing Hermes Programs

Name: `Parse Def`	Type: `initParse!initParseDefFn`
Name: `Parse Proc`	Type: `initParse!initParseProcFn`

These functions are used to create processes which can parse Hermes programs and return them in their abstract representations. To create a definitions module parser, for example, the `Parse Def` C-Hermes capability is retrieved, and an initialization call is made on it, passing it a collection of capabilities it will need for loading and storing Hermes object files, etc. A capability is returned that can be called with a Hermes definitions module (the source text, as a character string) and module name, and returns a number of structures resulting from the parsing of that module, including the value that should be stored in an object file (a `.do` file), various mappings between various id's (nominals) and the textual names appearing in the source file, and a list of errors. The `Parse Proc` service operates similarly.

These functions are used by the Hermes compiler programs (`dcom` and `pcom`), and are not expected to be used directly by normal Hermes programs.

5.2.8 Real Number Conversions

Name: `crtoa`	Type: `crtoa!ConvertRealToAsciiFn`

be used to make a linking call to a process on another interpreter. The output port that is returned is a copy of the output port that was provided in the `set linking port` operation on the target interpreter. Becuase the `generic` interface conveys polymorphs, the caller and callee can exchange whatever information they choose, including additional ports that will then operate across the network.

A program that provides a capability to print on some printer to other interpreters around the network might include code like the following:

```
declare
  gQ: genericQ;
  gFn: genericFn;
  request: generic;
  printer: printFn;
  ...
begin
...
unwrap startNet from CLoad("start network") {init};
call startNet(interpNo);
new gQ;
connect gFn to gQ;
unwrap setLink from CLoad("set linking port") {init}
call setLink(gFn);
while ('true')
  receive request from gQ;
  wrap copy of printer as request.value;
  return request;
end while;
...
```

Correspondingly, a program on another host that prints on the printer might look something like this:

```
declare
  link: genericFn;
  printer: printFn;
  request: polymorph;
  empty: empty;
  ...
```

5.2.5 Current Date and Time

Name: `date and time` Type: `time!getCurrentTimeFn`

This capability returns information about the current date and time, expressed in a number of convenient ways. See the definitions module `time.d` to see exactly what is returned by this call. A simple way to print the current date and time is:

```
...
unwrap getTime from CLoad("date and time") {init};
currentTime <- getTime();
call args.terminal.putLine("It is now " | currentTime.formattedTime);
...
```

5.2.6 Network Operations

Name: `start network` Type: `distributed!startNetworkFn`
Name: `set linking port` Type: `distributed!setLinkingPortFn`
Name: `get link` Type: `distributed!getLinkeFn`

The current Hermes implementation supports a somewhat awkward facility for allowing two different Hermes interpreters to interoperate. Higher level support for these facilities is described in Section 7.

In order to interoperate with other interpreters, each Hermes interpreter must have a unique identity. We use a combination of a host address and an *interpreter number* for this purpose. The interpreter number is an integer, with the sole restriction being that no two interpreters running on the same host may have the same interpreter number.

The `start network` operation supplies an interpreter with its interpreter number. Once this has been assigned, the interpreter is able to register itself so that other interpreters can establish communications with it. The `start network` operation can be performed only once during the lifetime of an interpreter. It must be performed before either of the other network operations can be performed.

The `set linking port` operation identifies to the interpreter an output port of type `distributed!genericFn`, to which incoming *linking calls* should be directed. Linking calls are the means by which other interpreters initially establish communication with this interpreter.

The `get link` operation returns an output port of type `distributed!genericFn` that can

These functions were provided explicitly for our window support, and their design is not very general. This will hopefully change at some point.

The `FD Read` and `FD Write` capabilities each take a file descriptor and return a capability that can then be used to read from or write to, respectively, that file descriptor. The file descriptor must have been set for asynchronous I/O. When an I/O request is made, the read or write process will attempt the operation and if the operation would block (that is, would cause the entire Hermes interpreter to block... not a good thing!), the C-Hermes process blocks instead, and a signal handler for a SIGIO Unix signal is established by the interpreter so the C-Hermes process can unblock and retry at a later time. Meanwhile, the Hermes process that initiated the read or write call will block waiting for that call to return.

The `FD Close` operation takes two file descriptor and closes them both. This will cause any processes waiting to read or write to those file descriptors to unblock and return with an exception. The fact two file descriptors are given instead of just one relates to the fact that this whole mechanism was designed to support asynchronous I/O to windows, and windows are always managed with a pair of pipes.

Notice that there is no facility for creating a file descriptor set up for asynchronous I/O. The `Window` service described above does this explicitly. This is yet another reason to generalize the asynchronous I/O support.

5.2.4 File Modification Time

Name: `File Modification Time` Type: `unixStat!statMtimeFunc`

This function takes a file name (relative path names interpreted according to the interpreter's working directory as a Unix process) and returns the file modification timestamp as an integer.

For example:

```
...
unwrap MTime from CLoad("File Modification Time") {init};
objFileModTime <- MTime(moduleName | ".po");
srcFileModTime <- MTime(moduleName | ".p");
if (objFileModTime < srcFileModTime) then
  -- recompilation needed
...
```

5.2.1 Functions For Which Hermes Wrappers Exist

Name: `Store Program`	Type: `objectio!writeobject_func`
Name: `Read Object`	Type: `objectio!readobject_func`
Name: `Write Object`	Type: `objectio!writeobject_func`
Name: `Unix`	Type: `unix!unixcall_func`
Name: `Access`	Type: `unix!access_func`
Name: `Read External`	Type: `unix!read_external_variablefunction`

The `"Store Program"` capability is not meant to be used by normal Hermes programs; it is used by the `"store"` resource described above. The other capabilities listed above are all available from standard resources listed in Section 5.1. Specifically: the `"Read Object"` and `"Write Object"` C-Hermes capabilities are available as the `"readObj"` and `"writeObj"` resources, respectively. The `"Unix"`, `"Access"`, and `"Read External"` C-Hermes capabilities are the `call`, `access`, and `r_external` components of the standard `"unix"` resource.

5.2.2 Windows

Name: `Window`	Type: `CWindow!CWindowFn`

This function provides some very rudimentary window support. It causes the interpreter to fork off a program that will manage a window for the purpose of displaying and reading text. The interpreter and the window program communicate via a pair of Unix pipes. Whatever is sent from the interpreter to the window program is displayed in the window, and whatever the user types into the window is sent up the pipe to the interpreter.

The call to create a window allows a label to be specified for the window's title bar, and also allows some SunView-style options to be specified, to determine foreground and background colors, sizes, etc. The Unix file descriptors for the interpreter's ends of the two pipes are returned.

Normally, you should not use this function directly. Higher-level support is described in Section 6.3.

5.2.3 Asynchronous I/O

Name: `FD Read`	Type: `fdhandlers!readInitFn`
Name: `FD Write`	Type: `fdhandlers!writeInitFn`
Name: `FD Close`	Type: `fdhandlers!closeInitFn`

5.1.7 User Name

Name: `username` Type: `common!charstring`

This resource can be requested from a "filtered" resource manager to find out what user name is being supplied on resource requests. When the root process creates a filtered resource manager, it looks for the environment variable `USER` to figure out a user name to use. If `USER` is undefined, it next tries `LOGNAME`. If that fails, it uses `"anonymous"`.

5.1.8 Standard I/O

Name: `stdin` Type: `stdio!stdin`
Name: `stdout` Type: `stdio!stdout`

These resources contain copies of capabilities appearing in the *terminal* component of the `main` interface. They are posted as resources, mostly because of the conventions used by programs meant to be run under the "piping shell" (see the `tools/pipe` and `tools/pshell` directories). It is unfortunate that there are two conflicting conventions for acquiring these capabilities. One or the other may disappear in future releases.

5.2 C-Hermes Functions

This section lists the capabilities currently available from the `CLoader` service described in Section 5.1.2. The functions will be described in much the same fashion as were the resources listed in Section 5.1. The CLoader is much like the resource manager in that it retrieves capabilities based on names supplied by the callers. Unlike the resource manager, there is no second argument required in these requests. As with the resource manager, the names supplied in the requests must be precisely as shown below; letter case and spaces are significant.

One current implementation bug is especially important for those who will be making direct calls to C-Hermes processes: these processes are not capable of returning the exceptions defined on their interfaces. When a C-Hermes process detects a situation in which a user-defined exception return should be made, its only recourse is to discard the callmessage, thereby causing the `discarded` exception associated with the call interface in the caller. In addition, C-Hermes processes are implemented in C, which is not a secure language. The implementations of these processes are generally fragile. We are currently working on better integration between Hermes and C, and when this has been achieved these processes will be made more robust.

Name: `writeObj`	Type: `objectIO!writeobject_func`
Name: `libWriteObj`	Type: `objectIO!writeobject_func`

These resources provide capabilities to store and retrieve arbitrary Hermes values in the Unix filesystem. They work much like the load and store capabilities described above, except they read and write values wrapped in polymorphs, rather than program values.

Since the Hermes prototype does not currently support persistance through fault-tolerance, you may wish to write an appointment scheduler that would maintain schedules in a file on disk called `"appointments"`. The following code fragment could be used to retrieve the value stored in this file into a Hermes variable:

```
unwrap pathReadObj from args.rm.get("pathReadObj", "") {init};
block begin
  unwrap appts from pathReadObj("appointments") {init}
on (readobject_func.file_not_found)
  -- no saved appointments... start with an empty calendar
  new appts;
on (polymorphMismatch, readobject_func.file_not_readable)
  -- unexpected exceptions
  ...
end block;
```

5.1.6 Current Working Directory

Name: `getCwd`	Type: `cwd!getCwdFn`
Name: `setCwd`	Type: `cwd!setCwdFn`

These resources allow an application to maintain the notion of a working directory that is separate from the working directory of the Hermes interpreter as maintained by Unix. Initially, the working directory is the same as the Unix working directory, but later `setCwd` calls can alter it, thereby changing the behavior of subsequent calls to `libStore` and `libWriteObj`. Note that each `setCwd` is interpreted relative to the interpreter's current workign directory as maintained by Unix. For example, if the interpreter is invoked while connected to the directory `/u/joe/programs`, then issuing `setCwd` with the argument `"foo"` would set the current working directory to `/u/joe/programs/foo`. A subsequent `setCwd` call with the argument `"bar"` would result in a working directory of `/u/joe/programs/bar`, not `/u/joe/programs/foo/bar`.

These resources are capabilities to load and store compiled program modules, using the Unix filesystem as a repository. Most Hermes programs will only ever need to load programs; programs that build other programs, like the Hermes compiler, use the store capabilities as well.

Technically, these interfaces should all mention the `checked` typestate attribute for the programs they load and store. This attribute is currently unimplemented, so it is left out of these interface definitions temporarily.[9]

The most basic load and store capabilites are called `"load"` and `"store"`. They should normally be used only with absolute pathnames, since relative pathnames will be interpreted relative to the directory from which the interpreter was started.

The more useful capabilities are `"pathLoad"` and `"libStore"`. The pathLoad capability searches all the directories listed in the `HPATH` environment variables for the requested program. Currently there is no easy way to change this path from within the interpreter, but by copying the relevant portion of the `hermes/root.pp` module, a new pathLoad capability can be constructed that follows a different path. The `"libStore"` capability resolves relative pathnames according to the current working directory as maintained by the `getCwd` process, described momentarily. The current working directory can be changed by a call to the `setCwd` capability described below.

An example of loading and running a program named `foo` from a file called `foo.po` somewhere in the `HPATH` environment variable is:

```
declare
  foo: fooFn;
begin
...
unwrap pathLoad from args.rm.get("pathLoad", "") {init};
foo <- create of pathLoad("foo");
call foo(...)
...
```

5.1.5 Reading and Writing Arbitrary Hermes Values

Name: `readObj`	Type: `objectIO!readobject_func`
Name: `pathReadObj`	Type: `objectIO!readobject_func`

[9] For this reason, the typestate checker currently does not complain when an unchecked program is instantiated with a `create of` or `procedure of` operation.

5.1.3 Unix Services

Name: `unix` Type: `unix!unix`

This resource contains capabilities to a smattering of Unix library and system call services. It is a record containing four pieces, as follows:

- *stdio*—This is a record containing a few dozen capabilities to specific I/O related Unix services. They are listed in section 5.3 below. The type of this component is `unix!stdio` and its wrapped typestate is {`full`}.
- *access*—This is a capability to perform `access` Unix system calls. It is of type `unix!access_func` and its wrapped typestate is `init`. You could use it to check whether a particular file is writable, for instance, with the following code:

```
...
unwrap unix from args.rm.get("unix", "") {init, init(access)};
if (unix.access("/etc/passwd", 'write')) then
  ...
```

- *r_external*—This is a capability to read certain global external variables in the interpreter (i.e. the Unix process in which the interpreter is running). These variables are: `errno`, `stdin`, `stdout`, and `stderr`. The *r_external* component is of type `unix!read_external_variablefunction`. For example, to find out why an open file operation failed, the following code might be used:

```
unwrap unix from args.rm.get("unix", "") {init, init(r_external)};
result <- unix.r_external('errno');
reveal result.int;
errno <- result.int;
```

- *call*—This is a capability to make a call to one of the supported Unix services. Normally, this capability is not used directly; instead, one of the capabilities in the *stdio* component is used. The type of this component is `unix!unixcall_func`.

5.1.4 Program Loading and Storing

Name: `load` Type: `load!load_func`
Name: `pathLoad` Type: `load!load_func`
Name: `store` Type: `load!storeFunc`
Name: `libStore` Type: `load!storeFunc`

listed here is {full}, which in most cases translates directly into {init}. As mentioned above, it is possible to unwrap these resources into typestates that are lower than their wrapped typestates.

5.1.1 Environment variables

Name: environ Type: root!environ

This resource is a table of pairs of character strings, corresponding to the Unix environment definitions in place when you started the Hermes interpreter. For example, if you wish to examine the value of the PATH environment variable, your hermes program might look like this:

```
...
unwrap environ from args.rm.get("environ", "") {init};
inspect pathentry in environ["PATH"] begin
  path := pathentry.value;
end inspect;
```

5.1.2 C-Hermes Processes

Name: CLoader Type: CLoad!CLoadFn

There are a number of "processes" that are implemented in C and made available by the interpreter itself. Capabilities to these so-called "C-Hermes" processes are obtained by means of the CLoader capability, which in turn is obtained from the resource manager. For example, if your program needed to obtain the current time of day, you would need access to the "date and time" C-Hermes process. Your code might look something like this:

```
...
unwrap CLoad from args.rm.get("CLoader", "") {init};
unwrap time from CLoad("date and time") {init};
call time(...);
...
```

The C-Hermes processes are listed and described in Section 5.2.

When specifying typestates on `unwrap` statements, remember that the current implementation does not support the `full` abbreviation except in definitions modules. This has the unfortunate effect of sometimes requiring lengthy typestate specifications on `unwrap` statements, as in:

```
unwrap unix from args.rm.get("unix", "") {
  init, init(call), init(access), init(r_external), init(stdio),
  init(stdio.stdin), init(stdio.stdout), init(stdio.stderr),
  init(stdio.clearerr), init(stdio.ctermid), init(stdio.cuserid),
  ...
  };
```

In such cases, one useful technique is to include only those portions of the typestate specifications that are actually required for your program to compile. In the above example, if the program were only going to make use of the `unix.stdio.clearerr` capability, then the following `unwrap` statement would suffice:

```
unwrap unix from args.rm.get("unix", "") {
  init, init(stdio), init(stdio.clearerr)};
```

Because the resource is actually fully initialized when provided by the resource manager, the above code fragment would cause most of the record to be coerced away at runtime as a result of the unwrap operation.[8] If the program never references these portions of the record, this coercion will cause no problems.

5.1 Standard Resources

This section describes the resources that are currently made available to Hermes main programs. All of these resources are currently given without restriction to any process that requests them from the resource manager. For each resource, we give the resource name, its type, and a short description. The typenames are all given in *modulename!typename* form. For example type `foo!bar` is the type `bar` defined in definitions module `foo`; you would look in the file `interface/foo.d` to see the type definition. Note that resource names must be given *precisely* as shown; case is important. The wrapped typestate of every resource

[8] Actually, this effect is not yet implemented; the record will be left fully initialized by the current implementation, though the typestate checker will treat it as if it had been coerced down.

- `putLine(s)` prints the string `s` followed by a newline.
- `getChar(c)` returns the next character typed by the user in `c`.
- `getString(s)` returns the next line of text typed by the user (without the terminating newline character) in string `s`.

The `terminal` argument is of type `terminalFunctions` as defined in module `terminalIO`. This module also defines all the interfaces for the individual terminal functions.

- *rm*—A *resource manager* that provides access to many other useful capabilities and values. The resource manager facility is described fully in Section 6.1. We identify the current set of resources provided by the system to main programs in Section 5.1.

The shortest Hermes *hello-world* program is as follows:

```
main:  using(main) process(Init: mainQ)
  declare
    Parms: main;
  begin
    receive Parms from init;
    call Parms.terminal.putLine("Hello, World!");
    return Parms;
  end process
```

The resource manager always returns resources wrapped in polymorphs. You must unwrap the resource in order to make use of it. For example:

```
foo: using (main, load) process (init: mainQ)
declare
  parms: main;
  pathLoad: load_func;
  bar: barFn;
  ...
begin
  receive parms from init;
  unwrap pathLoad from args.rm.get("pathLoad", "") {init};
  bar <- create of pathLoad("bar");
  ...
end process
```

will be cached. For instance, when compiling many modules from the cached shell, the compiler will be loaded on only the first compile. This dramatically improves compilation time on subsequent compiles.

5 The Standard Main Interface

When a Hermes program starts up, the Hermes run-time system provides initial capabilities (e.g., to load in additional modules, to print to the terminal, etc.) to the program. This is done by making a call to the program on its initialization port. The interface for this call is the `main` interface, and is defined in the file `interface/main.d`.

The main interface comprises the following types: `main!main` (the callmessage type), `main!mainQ` (an inport for receiving `main!main` callmessages), and `main!mainFn` (an outport for sending `main!main` callmessages). Any process that you wish to run as a main Hermes program (e.g. from the shell) should import the `main` definitions module and declare its initialization port to be of type `mainQ`.[7]

All definitions modules referred to in the following can be found in the `interface` directory in the Hermes installation.

There are three components to the main interface:

- *argv*—An ordered table of character strings, comprising the words typed on the command line that was used to invoke the program. An additional word, conventionally the name of the invoking program, is prepended to this list. Thus if you type `foo 1 hello` to the shell, the `foo` program will receive four strings in its `argv` argument: `"shell"`, `"foo"`, `"1"`, and `"hello"`. The `argv` component is of type `charstringList` as defined in the definitions module `common`.
- *terminal*—A record containing five capabilities (output ports) that can be used to perform input and output to the user's terminal:
  - `putChar(c)` prints the character `c`.
  - `putString(s)` prints the string `s`.

[7] In general, we encourage adopting the following naming conventions: if an callmessage type is named `foo`, then an inport type for receiving `foo` messages is `fooQ` and an outport type for connecting to `fooQ` inports is named `fooFn`. Much old Hermes code in the current distribution uses various different conventions, sometimes even within a single definitions module!

The easiest way to operate in this mode is by starting hermes via the command **hsh**. You can run the same hermes commands, with the same arguments, from the resulting prompt, as you can from the Unix prompt via the **hermes** command described earlier. The differences is that the interpreter need be started only once, and Hermes object files are each loaded only once, even if they are requested many times. For example, to compile process module `foo`, you can use either of the following:

```
hermes pcom foo
```

– or –

hsh

```
shell:  pcom foo
```

In the second form, "`shell:`" was printed by the Hermes shell program.

The `hsh` command is an abbreviation for `hermes cache shell`. The `cache` can be ignored for now; it is explained below.

The `shell` is a program that is much like the shells you use to execute Unix commands (`/bin/sh`, `/bin/csh`, etc.) It enters a simple loop in which it accepts a command typed at its prompt, and then executes the command, just as if it had been typed from the Unix shell as command line arguments to the `hermes` command.

Besides running Hermes programs, the shell supports a limited number of built-in commands. To change directories from the shell, issue the command `cd` *pathname*, where *pathname* is a complete Unix path. For example, to connect to `/distributed/computing/lib` from `/distributed/computing`, issue the command "`cd /distributed/computing/lib`." The `pwd` command displays the current directory, and `exit` causes the shell to terminate.

Another utility provided with the Hermes system is a software cache. It is invoked by a command like "`hermes cache foo`," where `foo` is a Hermes program. This will run the program `foo` , and will also cache all Hermes object modules loaded during the execution of `foo`. Hence, if `foo` requests that process `bar` be loaded many times, only the first request will result in `bar` actually being fetched from disk. Subsequently `bar` will be loaded directly from the cache (assuming that the file `bar.po` has not been changed since it was first loaded). For many applications, the cache can significantly improve run-time performance.

The best way to use the Hermes shell is in conjunction with the cache by invoking the command "`hsh`," which is an abbreviation for "`hermes cache shell`." This will create a shell with a software cache. Any modules loaded while running programs from the shell

to compile the definitions module `foo`, issue the command, "`hermes dcom foo`." `dcom` will look for the file `foo.d` in the current working directory. If you want to compile a definitions module whose source file is in some other directory, you'll need to specify a more complete pathname to as the argument to `dcom`, as in "`hermes dcom /hermes/defs/foo`." In either case, the compiled definitions module will be written to a file `foo.do` in the current directory.

The compiler `dcom` has four phases: parsing, checking, and two types of typestate rewriting.[5] A successful compilation of definitions module `foo` will generate the following output:

```
parse[foo.d] fixabbrevs checkdefs shorten store[foo.do]
```

Hermes process modules are compiled by the program `pcom`, which operates very much like `dcom`. The compiler `pcom` has four phases: parsing, type checking, typestate checking, and code generation. A successful compilation of module `bar` will generate the following output:

```
parse[bar.p] typecheck typestate codegen store[bar.po]
```

Both `dcom` and `pcom` may be given multiple module names on a single command line; all the specified modules will be compiled in the order given. If either `dcom` or `pcom` detects any errors in a module, it will print one or more error messages and then continue with the next file.

4 Using the Hermes shell and cache

Executing every Hermes command from the Unix command line performs poorly because Hermes must reads many object modules from Unix files. This is true even if a very simple program is executed, since the initialization perfomred by the root process requires the loading of many modules before the user's program can be run.[6] To improve performance we provide the Hermes shell and cache, which together provide an environment in which you can efficiently load and run many Hermes programs. We recommend that you use the cache and the shell.

[5] The typestate rewriting phases convert certain typestate constructions into the canonical form required by the typestate checker.

[6] You can see exactly what modules are required during initialization by setting `HDEBUG` to 2 or greater, watching all the load messages that are produced before your program is loaded.

currently executing. In addition, if another process were to turn tracing off, tracing of this process would stop as well.

2.3.8 HPROF, HPROFFILE, and HPROFINTERVAL

A rudimentary profiling facility is available for Hermes. If the HPROF environment variable is set to the value 1 or 2, the interpreter will collect profile information about Hermes processes as they are run. If the HPROF level is 1, every single instruction is counted by the profiling support. If HPROF is 2, a timer-based sampling scheme is used; every HPROFINTERVAL microseconds (100,000 if HPROFINTERVAL is not defined), the interpreter looks to see which instruction of which process is currently executing, and bumps that instruction's frequency count. When the interpreter terminates normally (not if it is interrupted, e.g. because you type ⟨control⟩C at the terminal), it writes the profile data to the file named in the HPROFFILE environment variable, or to hprof if HPROFFILE is not set.

There are two Hermes programs that can help interpret profiling data: profdetail and profsum. The former displays individual instruction counts for each instruction of every process, while the latter simply shows totals for each process. To invoke profdetail, issue the command profdetail *prof-file* either to the Hermes shell or on the command line when the interpreter is invoked; *prof-file* is the name of the file containing the profile information, and defaults to hprof if omitted. hprofsum is invoked similarly.

3 Compiling and Executing Hermes Programs

To run a Hermes program bar (stored in file bar.po) from a Unix shell, issue the command:

hermes bar *args* ...

hermes is a shell script that invokes the Hermes interpreter. You may need to check with your system administrator to find out where the shell script has been installed, and add that to your Unix PATH environment variable.

The command, "hermes bar *args* ..." starts a root process that parses the command line and attempts to load and execute the process module bar. Section 5 below discusses how the command line arguments *args* ... can be retrieved by program bar.

The Hermes program that compiles a Hermes definitions module is called dcom. Hence

2.3.7 HTRACE

The HTRACE environment variable can cause LI instructions (the "assembly" instructions of the Hermes interpreter) to be traced as they are executed. You will need to understand a great deal more about the interpreter than can be explained in this document in order to understand trace output in detail, but trace level 5 can be useful even without a detailed understanding.

There are currently two trace levels that are important to the system: 5 and 10. Level 5 shows only operations involved in message communication and process creation and termination. Level 10 shows all instructions, and typically results in far more output than anybody will ever want to see.

Level 5 trace output can be useful to see the sequence of communications among the processes in your program. The name of the process issuing the traced instruction is shown at the far left of the trace line, and the instruction display is indented according to the current "call level" of the executing process. A few factors make it somewhat difficult to interpret this output in some situations. First, since many processes may be runnable at any given time, instructions from several processes may be interleaved in the trace output. Attempting to track a single process can be difficult. In addition, multiple instantiations of the same program will be traced with the same process name and will therefore be indistinguishable in the trace output. Despite these problems, trace level 5 is sometimes the most effective means of tracking down a problem with a Hermes program. We hope to develop more sophisticated tools in the future.

Tracing can be selectively turned on and off under program control as well, by means of "pragmas" attached to statements. In particular, the following will cause level-5 trace output only during the execution of the given block:

```
...
pragma "trace=5"
block begin
  ...
end block;
pragma "trace=0"
...
```

Note, however, that the trace level is global throughout the interpreter. The above would cause tracing not only of the process in which it appears, but also of all other processes

2.3.4 HALTSYS

Normally, the `hermes`, `hparse`, and `hsh` shell scripts will attempt to run binary executable programs of the same names in the directory *hermes-root*`/mbin/`*arch*, where *hermes-root* is the directory where Hermes is installed on your system, and *arch* is a name for your machine architecture (normally, whatever is printed by the `arch` command on your system). If `HALTSYS` is defined, its value is appended to this directory when looking for the executables. For example, if your system administrator has installed a set of optimized executables in the directory *hermes-root*`/mbin/`*arch*`/opt`, you could set `HALTSYS` to `opt` in order to use the optimized programs rather than the standard set.

2.3.5 HROOTPROG

When the Hermes interpreter starts up, it initially creates a "root" program that is responsible for interpreting command line arguments and spawning whatever other Hermes processes are required. The root process expects, on its initialization port, a message of type `root!rootInitMessage`.[4] By default, the root process is loaded from the file *hermes-root*`/bin/root.po`, regardless of the setting of the `HPATH` variable. If you wish a different root process to be run, specify its complete path name in `HROOTPROG`.

2.3.6 HLOADPATH and HLINKPATH

There are actually two additional environment variables besides `HPATH` that can affect which directories are searched for program modules. If `HLOADPATH` is set, it is used in preference to `HPATH` for the loading of *program modules only*. If `HLINKPATH` is set, it is preferred over both `HLOADPATH` and `HPATH` for program modules that are statically linked (i.e. they appear in the `linking()` list of a program module being compiled). `HLINKPATH` will not affect the search for program modules loaded dynamically via `pathLoad`, and neither `HLOADPATH` nor `HLINKPATH` will affect the search for definitions modules or other Hermes values loaded via the `pathReadObj` facility.

`HLOADPATH` and `HLINKPATH` are required primarily for certain phases of the Hermes bootstrapping procedure. It is not expected that they will be required for normal use.

[4] Note that this is not a callmessage; the root process should not attempt to return it.

1. - If an unhandled exception occurs (this should never happen for programs parsed by the Hermes compiler), the offending process's status info is displayed.
   - An error message is displayed when a Hermes object file cannot be opened for reading.
   - An error message is displayed when a second attempt is made to set the network linking port.

2. A message is displayed every time a Hermes object file is read. This can be very useful for tracking down problems related to incorrect HPATH settings, since the full pathname of the object file is displayed. The message says `XDR Reading '<full pathname>'`.

7. All messages sent to or received from remote interpreters are logged.

15. - When an `OTHERS` handler is invoked for a process, the process' status is displayed as described below.
    - A message is generated when an interface mismatch occurs. The message includes the names of the processes involved in the attempted create operation, as in:

      Interface mismatch: 'foo' can't create 'bar'

20. `malloc_debug(1)` is issued, to cause extra checking to be performed by `malloc` and related routines. (*Sun only*)

99. `malloc_debug(2)` is issued, to cause the entire heap to be checked for consistency on every call to `malloc` and friends. (*Sun only*)

2.3.3 HROOTDIR

If this environment variable is set, it should identify the directory where Hermes is installed in your system. If it is not set, Hermes will attempt to look up a user named "`hermes`" on your system and use that user's home directory.[3] If neither of these situations holds, the Hermes interpreter cannot run.

If your system administrator is unable or unwilling to create a user ID named "`hermes`", then you will need to set the `HROOTDIR` environment variable in order to run Hermes.

[3]System administrators: It is not necessary for this ID to have a valid password or login shell.

are created, and how customized versions that use different path specifications could be constructed.)

2.3 Other Unix environment variables

This section lists assorted Unix environment variables in addition to `HPATH` that are meaningful to the Hermes system. Like `HPATH`, they are global to all processors running in a single interpreter, and cannot be altered once the interpreter has started.

2.3.1 `BLOCKINGIO`

The `BLOCKINGIO` environment variable determines whether the interpreter blocks for terminal IO. If BLOCKINGIO is set (its value is irrelevant), then the interpreter will block whenever any process requests input from the terminal. If BLOCKINGIO is not set, then the interpreter will not block when waiting for I/O. The disadvantage of using blocking I/O is that the whole system (all the processes running in a given interpreter) will hang whenever one process requires input. The disadvantage of using non-blocking I/O is that it is currently implemented using polling and therefore degrades system performance.

Note: The implementation of non-blocking terminal I/O is currently very buggy. We recommend that you always leave your BLOCKINGIO environment variable set when running Hermes programs. We hope to fix non-blocking I/O and redesign it so that polling is not required in the near future.

2.3.2 `HDEBUG`

The `HDEBUG` environment variable is an integer that controls the level of debugging information displayed by the Hermes interpreter. The higher the value of `HDEBUG` the more debugging information displayed. Higher `HDEBUG` levels always imply all lower levels. Currently defined debugging levels are as follows:[2]

They are special in that they must be run through the C preprocessor program (cpp) before being compiled. This processing is necessary for many of the modules that are built early in the Hermes bootstrapping process, which is a complex procedure that builds a complete Hermes system from nothing but source files.

[2]These are very likely to change.

module, respectively. Thus, compiling the source file "`foo.d`" would result in the creation of a file "`foo.do`", and compiling "`bar.p`" would produce "`bar.po`".

The compiler always writes object files into the "current directory," which is normally the directory in which the Hermes interpreter was started (but see the `setCwd` service described in Section 5.1 for how to change this).

2.2 Where Hermes looks for files

To compile a module that imports another module, the compiler must read the imported module's object file. For instance, to compile a module `bar` with `foo` in its `using` list, the compiler must load the compiled definitions module `foo.do`. Similarly, if process module `boz` statically links module `zip`, then the compiler must load the compiled process module `zip.po` in order to to compile `boz`. *This implies that imported or statically linked modules must be compiled before any module that uses them.*

It is also sometimes necessary for compiled modules to be loaded during the execution, rather than during the compilation, of a program. This is because programs are supplied with facilities for dynamically loading programs and instantiating them into running processes (see the description of the `pathLoad` resource in Section 5.1).

Frequently, the modules that must be loaded either to compile or to execute a Hermes program are to be found someplace other than the current working directory. They may be standard modules that are provided as part of this Hermes system, or they may be modules written by other individuals and kept in their own directories.

To accomodate these requirements, the compiler (and the standard resources `pathLoad` and `pathReadObj` (see Section 5.1)) will search all of the directories specified in the Unix environment variable `HPATH` when looking for modules to load. The `HPATH` variable, like the standard Unix `PATH` variable, should be a colon-separated list of directory path specifications. The default `HPATH` value, if you leave it undefined, is `.:`*hermes-root*`/bin`, where *hermes-root* is the directory in which Hermes has been installed on your system. Paths beginning with "`~/`" or "`~`*user*`/`", commonly understood by Unix shells, are not understood by Hermes.

The `HPATH` environment variable is "global" for all processes running in an interpreter and cannot be changed after the interpreter has started. (But look in the source file of the `root` process, in file `hermes/root.pp`,[1] to see how the `pathLoad` and `pathReadObj` resources

[1]This process module and others in the Hermes distribution have a file extension of `.pp` rather than `.p`.

- `!K windowname` - kill `windowname`
- `!Q` - quit window system

This is obviously not a realistic window manager, but its structure can be used as a guide to building something more practical. You should try it with multiple windows, all three of the tutorial's standard interface programs, and tickle its error conditions. Why doesn't the front-end prompt always come out before you start to type in the next command?

Suggested exercises: Write a `standardIn` program that uses the tokenizer to count characters words and lines like the Unix wc command. Run this program with `compound`. Write a more robust `boundedbuffer`. This requires scrapping the GetLine and PutLine interfaces and defining GetBuffer and PutBuffer interfaces for the buffer. The drawback of this change is that this `boundedbuffer` will no longer compose with other processes that interface with PutLine and GetLine.

2 Using Hermes under Unix

2.1 Storing Hermes modules in Unix files

The Hermes language includes notions of definitions and process modules. In a fault-tolerant, non-terminating Hermes system, all Hermes data would persist until discarded, and there would be no need to explicitly save modules to disk storage. The current implementation is neither fault-tolerant, nor is it run for long periods of time, typically. For this reason we use the Unix filesystem as a repository for Hermes program modules and other data values. This requires a set of well-defined rules for naming the files that contain Hermes data, as well as facilities for loading and storing data. This section describes the naming conventions assumed by the compiler and run-time system of the current implementation. Section 5.1 discusses the services available to the Hermes programmer for managing Hermes data in the Unix filesystem.

Hermes programs consist of definition modules and process modules. In their source forms, these files should be named with extentions "`.d`" and "`.p`", respectively. For example, the source text for a definitions module named foo should be placed in a file named "`foo.d`". Likewise, the source for a process module named "`bar`" should go in a file called "bar.p".

When a Hermes source file is compiled, the compiled object form is saved in a file with the same base name, but with the extension "`.do`" or "`.po`", for a definitions or process

`Compound` is especially interesting because it loads another program and connects that program's output to `filter`. Any program that has a `standardIn` initialization port will work, including `compound` itself. You might try running `hermes stdintf compound helloworld/`, `hermes stdintf compound echo/`, `hermes stdintf compound compound/echo/`, etc. You might also write another `standardIn` program and make it a client of `compound`.

The `boundedbuffer` program is a server program—it has an infinite loop in which it receives and services calls. The main program `bbintf` is an interactive front-end that lets you be the client of `boundedbuffer`. Enter `hermes bbintf boundedbuffer`. `bbintf` responds to three commands

`P line` puts `line` into the bounded buffer

`G` gets the first entry from the bounded buffer

`Q` quits the bounded buffer program

`boundedbuffer` also accepts an optional argument, the size of the buffer. Try `hermes bbintf boundedbuffer 8`.

You may notice that `hermes` terminates when you try to put into a full boundedbuffer or get from an empty boundedbuffer. It terminates because there are no processes ready to run, so the interpreter has nothing left to do. When you type at the `bbintf` prompt, `bbintf` is doing a `getline` and is the only runnable process. It then makes a get or put call to `boundedbuffer` and blocks, waiting for the return. If `boundedbuffer` cannot handle the call, because doing so would exceed its buffer capacity, then it cannot run either. Thus there are no runnable processes, and the interpreter terminates. To design a more robust boundedbuffer you might want to provide exception returns on the put and get calls, so an unsuccessful call terminates.

The window system is the last and most elaborate tutorial example. To run it enter `hermes stdintf wmsystem`. Its built-in front-end understands the following commands:

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

- *Value constructor.* A new C program program, "construct," can be used to construct arbitrary Hermes values and store them in files in a form suitable for reading via the "readObject" and "pathReadObject" functions.
- *Name changes.* The "hermi" command has been renamed to "hermes", and what was formerly called "hermes" is now "hparse." There's also a new command "hsh" which is simply an abbreviation for "hermes cache shell."
- *New platforms.* Ports to three new platforms have been provided by users of release 0.6alpha:
  - NeXT Machine: ported by Seth Strumpf and Rob Lehman of Columbia University's Center for Computing Activities.
  - HP 9000-series running HP-UX: ported by Willard Korfhage of Brooklyn Polytechnic Institute.
  - Convex: ported by Willard Korfhage of Brooklyn Polytechnic Institute.

1 Working through the Tutorial

We recommend that you run the tutorial code. Source files for the tutorial code examples are stored in the directory `hermes/tutorial_examples`. Many of the tutorial examples are not complete Hermes modules in the Manual—some include only the code and others only the declarations. All these examples have been merged into single files that compile and execute in the on-line examples.

Most of the tutorial example programs receive their initialization message on a `standardIn` input port (except `boundedbuffer` whose initialization port is a member of the type `bbIn`). Like all languages, this implementation of Hermes defines a *main* program, which is the kind of program that the interpreter can load and run directly. A Hermes main program has an init port of type `mainQ` (see Section 5 for more details) which passes it capabilities to many Unix and Hermes facilities.

To run the tutorial examples use a small main program called `stdIntf` which will load a tutorial program that uses the `StandardIn` init port and connect it up. (To run the `boundedbuffer` program use the main program `bbintf`.) `stdIntf` also takes the rest of the character string input and passes it to the tutorial program. For example, to run the `helloworld` program, enter `hermes stdintf helloworld`. To run the `echo` program, enter `hermes stdintf echo`.

0 Change History

0.1 Prior To Release 0.6alpha

This User's Guide was first prepared for release 0.6alpha, which was also the first release that was made available for outside users. There is no change information from prior releases.

0.2 Changes From Release 0.6alpha To Release 0.7alpha

- *Real numbers.* Some rudimentary support for real numbers has been added. Specifically, machines with a 32-bit int and a 64-bit double float should work properly. The ACCURACY specification on real type definitions is ignored. Depletion exceptions are raised for real computations that exceed the range or precision of the representation.

- *Polymorph checking.* Polymorphs are now actually wrapped with correct type information, and this information is checked on UNWRAP operations. Currently, polymorphs are wrapped with empty typestates, and the wrapped typestates are still ignored on UNWRAP operations.

- *Improved MAIN interface.* The MAIN interface (defined in interfaces/main.d) has been substantially redesigned for added flexibility and convenience. Most of what was formerly provided in the main interface is now available via a resource manager that is provided directly in the main interface. The old main interface is available under the name "oldmain," and a program named "oldmain" translates the new main interface to the old for the purpose of running programs written to the old main interface. The oldmain definitions and process modules may disappear in some future release. The "rmain" interface, which was formerly an alternative to "main," has been removed; programs written to it should be converted to use the new "main" interface.

- *Unbundled distributed support.* The functionality that was formerly provided in the "dmain" program has been split into three separate programs: "startnet," "getlink," and "linkserver." See section 7 for details.

- *Definitions module checking.* Complete checking of definitions modules is now performed by "dcom." This includes detecting erroneous modules that cannot be produced by the Hermes parser but could nevertheless be constructed by a Hermes program.

6.2.2 Options 37

6.2.3 Delimiting the scope of makes 38

6.2.4 Bugs 39

6.3 Windows 39

7 Distributed Hermes **40**

7.1 The Truth About the `Main` Interface, the `Root` Process, and the `Cache` . . 40

7.2 The `StartNet` Program 41

7.3 The `LinkServer` Program 42

7.4 The `GetLink` Program 42

7.4.1 An Example 42

8 Unsupported Features and Known Bugs **43**

This user's guide describes how to use Hermes on Unix and overviews the software tools distributed with the Hermes system. Section 0 contains summaries of changes from prior releases of the system. Sections 2 and 3 describes the mapping between Unix files and Hermes modules, and explains how to use the Hermes compiler. Section 4 presents a convenient environment for running Hermes programs—the Hermes shell. Section 5 outlines *capabilities* provided to every root Hermes process. These capabilities enable a Hermes program to do I/O, invoke Unix commands and system calls, and use other Hermes library programs. Section 6 presents programming tools provided with Hermes, including a make facility, a windows package, a resource manager, and a set of services for executing distributed Hermes programs. Section 8 describes features of Hermes that are unsupported and bugs in the current release.

5.1.5 Reading and Writing Arbitrary Hermes Values 22
5.1.6 Current Working Directory 23
5.1.7 User Name 24
5.1.8 Standard I/O 24
5.2 C-Hermes Functions 24
5.2.1 Functions For Which Hermes Wrappers Exist 25
5.2.2 Windows 25
5.2.3 Asynchronous I/O 25
5.2.4 File Modification Time 26
5.2.5 Current Date and Time 27
5.2.6 Network Operations 27
5.2.7 Parsing Hermes Programs 29
5.2.8 Real Number Conversions 29
5.3 Unix Stdio Functions 30

6 Tools **30**

6.1 Resource manager 30
6.1.1 Using the Resource Manager 32
6.1.2 What About User Names 34
6.1.3 Creating Resource Managers 34
6.2 Make facility 36
6.2.1 Module dependencies and consistency 36

2 Using Hermes under Unix **8**

2.1 Storing Hermes modules in Unix files 8

2.2 Where Hermes looks for files 9

2.3 Other Unix environment variables 10

2.3.1 `BLOCKINGIO` 10

2.3.2 `HDEBUG` 10

2.3.3 `HROOTDIR` 11

2.3.4 `HALTSYS` 12

2.3.5 `HROOTPROG` 12

2.3.6 `HLOADPATH` and `HLINKPATH` 12

2.3.7 `HTRACE` 13

2.3.8 `HPROF`, `HPROFFILE`, and `HPROFINTERVAL` 14

3 Compiling and Executing Hermes Programs **14**

4 Using the Hermes shell and cache **15**

5 The Standard Main Interface **17**

5.1 Standard Resources 19

5.1.1 Environment variables 20

5.1.2 C-Hermes Processes 20

5.1.3 Unix Services 21

5.1.4 Program Loading and Storing 21

Hermes: Unix User's Guide

— DRAFT —

Version 0.7alpha

Robert E. Strom

David F. Bacon Arthur P. Goldberg Andy Lowry
Daniel Yellin Bill Silverman

IBM T. J. Watson Research Center*

March 19, 1991

Contents

0 Change History **5**

0.1 Prior To Release 0.6alpha 5

0.2 Changes From Release 0.6alpha To Release 0.7alpha 5

1 Working through the Tutorial **6**

*PO Box 704, 30 Saw Mill River Road, Yorktown Heights, NY 10598