Hermes vs. Modula-3

Joel Auslander

December 7, 1994

1 Introduction

1.1 System Programming

Hermes and Modula-3 represent two different approaches to providing language support for systems programming. Systems programming is the programming of large, persistant software systems, such as operating systems or transaction systems. It is also distinguished by the issues of reliability, maintainability, security, and concurrency, which dominate in such large systems. These issues in lead to desirable characteristics for a systems programming language, such as: analyzability, modularity, error handling, synchronization, portability, static checking, incremental programming, and expressive interface specifications.

2 Modula-3

2.1 Origins

Modula-3 can trace its lineage back to the dawn of time: Algol begat Pascal begat Modula-2 begat Modula-2+ begat Modula-3. Each step in this evolution tried to improve on the imprevious generations, with the later languages also incorporating ideas from other language families. From Modula-2 on this family of languages has been moving towards support for systems programming. Modula-2 already included concepts useful for systems programming, such as modules and a concurrency model (coroutines). The languages Modula-2+ and Modula-3 were developed by Digital's Systems Research Center and continue this evolution. Modula-3 already has a several decent compilers, and is being seriously used. For example, the SPIN project here at University of Washington has basically settled on Modula-3 as its implementation language.

2.2 Modula-3 at a Glance

Modula-3 does not introduce any new concepts, but instead consolidates existing concepts that are already widely used in languages such Modula-2, Ada, C++, and Lisp. Where Modula-3 innovates is in how cleanly these concepts coexist. While it has the features list of a "big language" such as Ada, it still maintains the simplicity and predictable behaviour of a "small language" such as Pascal. One example of this predictability is that there are (in the safe subset) no uncaught run-time errors: there are no holes in the static typechecking, no way to use deallocated storage, and no direct pointer manipulation. The closest Modula-3 comes to innovation is where it combines two closely related concepts by supporting both rather than choosing one. For example, it has both abstract data types and objects, it has both garbage collected and non-garbage collected storage, and it has both safe and unsafe modules, where the safe modules are restricted to use the safe subset of the language.

Another way Modula-3 cleanly combines ideas is to separate out the program concepts to allow a subset of them to be used independently. By contrast, a language that combined concepts might define passive data to have have exposed structure and be stack allocated and objects to have hidden structure and be garbage collected. Modula-3 separates out the concept of garbage collected vs. non-garbage collected values from the concept of objects vs. passive data from the concept

of hidden vs. exposed structure. One can define an object whose internal structure is visible and which isn't garbage collected, or a passive data type which is garbage collected and whose structure is hidden, or any other combination. Modula-3 even distinguishes between information hiding and name-based type equivalence. Therefore it is possible, although not recommended, to define an abstract type whose structure is hidden by the interface, such that untrusted modules that correctly "guess" structure can reference its internals. Conversely, you can have a type where its structure is visible but where only trusted modules can properly name and access this structure.

When language concepts do interfere, Modula-3 is not above ad-hoc fixes. For example, allowing variables to store (references to) procedures conflicts with the scoping rules, because a nested procedure could be bound and then called after its stack frame disappears. Modula-3 fixes this by only allowing top-level procedures to be assigned to variables, but allows nested procedures to be passed as parameters (which is safe). As another example, assigning a value with a supertype to a variable with a subtype is normally allowed and generates a run-time check. This check is possible because most values are tagged with their true (allocated) type. This type of assignment is legal for traced passive types, traced object types, and untraced object types. Untraced passive types, however, are represented just as raw data and are not tagged with type information. Modula-3 deals with this by adding the ad-hoc restriction that such assignments are illegal for untraced passive types.

The basic list of Modula-3 features is

- built-in passive data types (scalars, arrays, records)
- abstract data types
- type hierarchy
- objects (methods, inheritance, polymorphism)
- garbage collected storage
- modules (structured namespace, hiding)
- generic types
- threads and synchronization
- exceptions

In addition, unsafe modules have acess to the following unsafe features

- explicit type loopholes
- non-garbage collected storage (explicit allocate/deallocate)
- pointer arithmetic

2.3 Modules

2.3.1 Interfaces and Implementations

As you'd expect, Modula-3 divides a program up into modules. There are two types of modules: definitions modules and implementation modules. Definition modules define an interface (all public types and functions). For every definition module there is a corresponding implementation module containing the code that implements the interface. This is the same approach taken by Modula-2 and a host of other languages. In Hermes definitions modules and code modules are separate: an arbitrary number of functions can implement the same interface. Under Hermes linking, your program itself decides which implemention to use, explicitly and dynamically.

2.3.2 Opaque Types

Modula-3's approach to information hiding is based on subtyping. An interface module, doesn't have to give all the information about a type–instead it can define a type as a subtype as another type. Instead of saying

```
TypeA = TypeB;
```

you can say only

```
TypeA <: TypeB;
```

This gives clients (modules who import the type) enough information to use it, as long as they restrict themselves to the operations that are valid on the supertype. This information hiding mechanism is used for both passive and object types. For passive types the hierarchy has only two levels: the only valid supertype for passive type is the generic pointer type `REFANY` (or the equivalent untraced type `ADDRESS`). Therefore for passive data this approach just allows a module to define abstract data types: types that will be treated by outside clients as pointers. The implementation module for an abstract data type will contains a `reveal` construct that defines the full structure of the abstract data type. For objects you can have a true type hierarchy, and can use this to define multiple interfaces to the same object which reveal different amounts of structure. A typical use is to define a public type which specifies the public methods, and then, inside the implementation module, `reveal` the objects full private structure.

2.3.3 Access Control and Branding

Modula-3 separates out the concept of access control from that of hiding. By access control I mean which modules can reference the internal structure of a hidden type. Just using the subtyping construction doesn't achieve any protection: anyone who "guesses" the internal structure can legal use a `reveal` to expose and access it. Modula-3 folds the issue of access with that of named vs. structural types: with a named type, you have to be able to specify the name to access it, with a structural type you need only know the structure. In both cases, Modula-3 uses the `brand` construct. This defines a type which is incompatible with others regardless of structure. In order to use a branded type you have to import the interface module in which the brand occurs. This can be used as the basis for access control: you put a branded full definition of a (passive or object) type in an interface module and control who can *import* it (how you control who can import what module is a detail of your compiler implementation or operating system). You also provide a public definition that anyone can import, but that hides the internal structure. Now your (external) access control over who can import what is reflected in the language in terms of who can expose what structure.

2.4 Generics

Modula-3 has generic modules. These are module constructures used for concepts like lists. A generic module takes formal parameters, like a procedure does. It then defines a module in terms of these parameterizing modules. For example, a generic binary tree module `Binary` might have a formal parameter `Element` and then define a binary tree interface in terms of of the base element type `Element.T`, and a comparison function `Element.compare`. A corresponding generic implementation model would implement the binary tree in terms of these same parameterized definitions. Then, if one had a module `String` that defined a type `String.T` and a function `String.compare` one could combine it with `Binary` to create a module that provided binary trees of strings. To keep things simple, the syntax requires that you give the combined module a new name, such as `StringBinary`. The files containing the interface and implementation of `StringBinary` define it as the instantiation of the generic module `Binary` combined with the parameter `String`. The result is just the same as if you did string replacement for the formal parameters in files that define the generic module `Binary`. Everywhere you want to import this composite module you import `StringBinary`.

Although Hermes doesn't have anything comporable, Hermes has the more power feature of letting programs dynamically manipulate code and instantiate that code as a running process. In

Hermes, therefore, one could implement something like this as a user level program, and even do such combinations dynamically.

2.5 Objects

The Modula-3 object system is very straightforward. Modula-3 supports single inheritance. Subtypes can override methods and add new instance variables. Subtypes can access overridden of their supertype by naming the supertype explicitly. This approach can be implemented simply and efficiently in terms of jump tables: a method table is associated with every object type, and an offset with every method that object type supports. This can be done so that the same method has the same offset for a subtype and its supertype. Every method reference is then compiled into a jump table reference indexed by the corresponding offset. At run-time each object is tagged with its type's jump table. The syntax for defining and using methods is also straightforward and traditional: `o.m(x)` is executed as a call `m(o,x);` where `m` is defined in the module that implements `o`'s (dynamic) type.

2.6 Memory Allocation and Aliasing

A Modula-3 program can refer to values stored in three regions of memory: the stack, a garbage collected heap, and a not garbage collected heap. The language knows statically which of these three regions any value is stored in, from the value's type. Non-reference types, such as arrays or scalars, are on the stack. Reference types (including object types) come in two flavours: traced and untraced. The programmer must explicitly define untraced types using the keyword `UNTRACED`. Traced reference types are stored in the garbage collected heap, untraced ones in the non garbage collected heap. The type rules prevent the programmer from mixing these types up by storing an untraced value in variable of a traced type.

Each region has its own allocation/deallocation scheme. Values on the stack are created when a scope is entered and destroyed when the scope is left. Values in the traced heap are created by the `NEW` command and garbage collected when there are no more references to them. Values on the untraced heap can only be created (by the `NEW` command) and never deallocated. Actually, untraced values can be explicitly deallocated by the unsafe operation `DISPOSE`. Untraced values are really intended to be managed outside of unsafe modules. They are used to share data between the safe Modula-3 world and the worlds of unsafe Modula-3, C, and machine code. Modula-3 programs that use no untraced references can conceivable run without a garbage collector. This is valuable for low level applications such as device drivers or embedded applications.

2.7 Exceptions

Exception handling is very straightforward: programs can define exceptions and raise them. Exceptions propage up the call stack until they find a dynamically enclosing handler. Exceptions that propagate past the top level terminate the program (which would be unimaginable in Hermes). Each procedure defines the exceptions it can propagate. Raising an exception in a procedure that can neither handle nor propage it also terminates the program.

Exceptions are meant for user defined conditions, and are not used to signal errors in the language itself such as out of memory, a reference through nil, or a run-time type error. There is no way to survive these errors: they terminate the program.

Modula-3 exceptions can also carry some data. The definition of an exception can (optionally) specify that it takes a single parameter. This parameter is then passed up to the exception handler. For example, you can pass an string containing an error message up to the handler. While this is nice to have, you might be able to do this almost as easily without a special language feature: since all exceptions are explicitly raised, you could store the error string in a global variable before raising the exception. Again, Modula-3 chooses completeness where Hermes chooses simplicity.

2.8 Threads

Modula-3 also provides support for threads. Most of this support is in the functions and data types defined by the module "Threads". In addition Modula-3 has added syntactic support for mutual exclusion locks (mutexes). The mutex data type itself is defined as the object type `Threads.Mutex`. It is used in the construct

```
LOCK mutex Do
 statement1;
 statement2;
 ...
END;
```

This code aquires the mutex before executing the body (blocking if necessary until the mutex is available) and releases it when the body completes.

The `Threads` module also supplies a function `Treads.fork` to create new threads. This function takes as its argument an object that inherits from type `Threads.Closure`. The newly forked thread executes this objects `apply` method. When the method returns, the thread terminates. The `Threads.fork` function returns a handle to the thread. This handle can later be passed to the function `Threads.join` which waits for the specified thread to complete and returns whatever value the thread's `apply` method ireturned

In addition, the Threads module implements condition variables, another type of synchronization object. The `Threads` module also provides an alternative function based interface to mutex locks.

2.9 Unsafe Operations

Almost all of Modula-3 falls into a subset defined to be safe. As long as you stay in this subset your program is guaranteed against uncaught run-time errors. Modula-3 also defines a few additional unsafe operations. These gives you direct access to addresses, address arithmetic, explicit type loopholes, and deallocation of untraced memory. Using these you can do anything.

3 Hermes

3.1 Origins

Where Modula-3 was produced by slow and steady evolution, **Hermes** almost sprang full-grown from the brow of its creature, Rob Strom. Hermes is actually the second version generation of the language, a successor to NIL. These languages have seen almost now practical use, except for a few proof-of-concept projects within IBM. This contrasts with Modula-3, which is a descendant of Pascal via Modula-2: Pascal is high in the top 10 of widely used languages, while Modula-2 and its cousins are also widely used through industry and academia. While Hermes is designed to be compilable to machine code (although not as easily as Modula-3) the current implementation of it compiles to an intermediate code which is interpreted.

Its hard to tell for sure whether anyone is using Hermes. However, while `comp.lang.modula3` contains lively discussions among Modula-3 users, `comp.lang.hermes` is dead.

3.2 Hermes at a Glance

Hermes is smaller, cleaner, and more abstract than **Modula-3**. It is also less conventional (or equivalently more innovative) than Modula-3, as well as less complete. Where **Modula-3** just combines features present in many traditional languages, Hermes is more radical.

Hermes can be excessively clever: the language components of processes, exceptions, and type-state knit together in complex and subtle ways. In Modula-3 the various language features are surprisingly independent. You can use just the subset you want and combine them in arbitrary

ways. As examples, Modula-3 has `TRACED` and `UNTRACED` reference types that can be object or passive and can be hidden or revealed and branded or not. Any of the 16 possibilities is valid. While in Hermes features combine cleanly because they are carefully interdesigned, in Modula-3 features combine cleanly because they are designed to be independent.

The following Hermes features stand out:

- Hermes completely throws out the ideas of pointers and addresses. By doing so, Hermes avoids any possibility of aliasing: any value can only be referenced one way. Hermes doesn't have circular data structures either.

- Hermes strengthens the traditional notion of types by adding a concept of *typestate*. Just as types allow static analysis to detect possible type errors, typestate allows static anlaysis to detect possible references through nil and possible misuses of variants.

- Hermes function interfaces express much more than Modula-3 signatures. An interface defines not only the types of the parameters, but whether they are initialized on input and on output, which parameters should be constant (unmodifiable within the function), what exceptions the function can raise, and what state the parameters are in after an exception. The interface specification allows user defined constraints to be added and checked. For example, a function can specify that it is only to be called on positive numbers, or that it always returns unit vectors.

- Hermes doesn't have a sophisticated type system. The values that variables hold and that processes pass around must be of a known, exposed type. There is no subtyping or abstract data types, as in Modula-3. Any hiding of information or implementation must be done by putting data completely inside a process. Some syntactic support is added for this, but the syntax still distinguishes between communicating with processes and manipulating local, passive data. There is a dynamic escape from the type system, the type polymorph, but using polymorphs requires additional syntax and run-time checks. For example, all polymorphs are the same, as opposed to Modula-3 which can recognize two abstract types as different without exposing their internal structure.

- Hermes replaces the traditional concepts of functions, modules, abstract data types, objects, monitors and threads with a single *process* abstraction.

- Hermes replaces traditional linking (through a global namespace) with explicit, programmer controlled dynamic linking. There is no global namespace in Hermes, unlike in Modula-3, where by naming a module you gain access to procedures and variables defined in it. A process (=program, module, function, etc.) must be explicitly passed any data or functions it needs. Whoever creates a process completely controls the environment that process sees. This fits with a *capabilities* style of access control.

- Because all Hermes processes commicate exclusively through message passing, Hermes can be directly implemented on a message passing distributed architecture. Implementing Modula-3 on such an architecture would effectively require the intermediate step of simulating shared memory via message passing.

- Hermes is much more abstract than Modula-3. Where Modula-3 can be implemented in a straightforward way using traditional compiler and system architectures, Hermes assumes an extremely intelligent compiler that can translate its abstractions (such as processes and tables) into more efficient implementations (such as function calls and arrays) while maintaining the proper semantics. The Hermes manual explicitly admits this tradeoff of easy of implementation and perhaps performance for a more abstract programming model.

- Hermes assumes the whole world is Hermes. Any communication between a Hermes program and the outside world must be done through specially written, trusted routines (currently implemented in C) that convert between Hermes semantics and C/Unix semantics. To contrast,

Modula-3 has the notion of an outside functions (that can be linked to through the `EXTERNAL` directive) and data (stored in `UNTRACED` references).

- Hermes is designed so that there is no distinction between a a function, a program, or an operating system. This makes sense because of the strength of Hermes encapsulation: multiple programs can run in the same world without being aware of one another, no matter how ill behaved or buggy some of the programs are. In Modula-3, on the other hand, all procedures share the same global name and address space, the module "Main" always gets control initially, and run-time errors terminate the program.

- Hermes builds the compiler into the language. You can create code on the fly and instantiate it. This makes Hermes more powerful, but requires a larger run-time system.

- Unlike Modula-3, Hermes doesn't include any loopholes that access the implementation underneath the abstractions. Further, all Hermes operations require the full run-time system, unlike Modula-3 in which modules that use only untraced references can be run without the garbage collector. Hermes is therefore impractical for writing low-level code: for example, the Hermes run-time system could not be implemented in Hermes.

- Hermes function call semantics (and exception propagation) fall almost magically out of the interaction of the different components. If you consider function calls as being more fundamental than processes or typestates, this approach is backwards. Modula-3, by comparison, defines function call semantics first and then designs everything else so that function calls work properly. For example, Modula-3 allows you to freely manipulate references only to *top-level* functions. This restriction makes the implementation of functions simpler (because it avoids issues of liveness) at the cost of ugliness in the language.

3.3 Processes

The most crucial Hermes concept is a process. A process is a self contained entity with its own private data (accessed through private variables), code, and a single thread of control. The data and thread are private: no two processes ever share the same data value or variable, nor can a single thread execute in two processes. For data Hermes maintains a stronger restriction that for any value there is a unique way of accessing it. Further, only one thread executes in a given process during that process's entire existance. The function call semantics sometimes make it seem as though a single thread is executing in two processes, but really two threads exist, one going to sleep and the other waking up in the places where it seems as if control is being "transfered." Since process has only one thread and private data, all data access is serialized.

3.4 Ports

Without any way for processes to interact Hermes would be a boring language: each process, trapped in its encapsulated environment, would be completely independent. However, because data isn't aliased, processes can't communicate through shared data. Hermes instead adds a second language contruct of a *port*. Ports come in two flavors: *inports* and *outports*. An inport can be connected to an outport, so that data be sent on the inport and can be read from the outport. Such a connection can exist even when the inport and outport reside in different processes. Ports are the only way processes ever interact: this includes the communication implicit in function calls and process creation. The semantics of both of these operations are defined in terms of ports.

3.5 Process Instantiation

Processes can be instantiated dynamically using a builtin operation `create of` *code*. The syntax implies that the creator has the code that the process will run. Creating a process simultaneously creates an outport (returned by the `create of` operation and an inport (which is accessible to the new process through a variable). Exposing this functionality allows Hermes to use explicit

programmer defined linking where standard languages (like Modula-3) define linking as part of the language. Also, the Hermes syntax allows the definition of code literals, which can be used in ways similar to the block syntax of smalltalk, and in ways similar to nested procedures in Modula-3.

3.6 Callmessages

In addition to the concepts of code, processes, and ports, Hermes adds the concept of a *callmessage*. Callmessages are aggregates that combine data with synchronization. They are created by the basic operation `call`, which assembles a callmessage from a list of parameters, puts the calling process to sleep, and sends the callmessage to another process (on a specified outport). Any callmessage is associated with the (sleeping) process that sent it. When another process executes `return` this process is reawakened, and the callmessage is dissassembled back into parameters. This corresponds to the *value-result* calling convention.

Because a callmessage is associated with the process that sent it, this sending process is guaranteed to be awakened when the callmessage is returned. Destroying a callmessage automatically returns it (and not only wakes the sleeping sender but raises an exception). A process can *forward* a call by passing the callmessage to another process. The behaviour of return only depends on the callmessage, so when a process returns a forwarded callmessage it returns directly to the sleeping process that sent it—the process that first received it is not involved in the return. A process receives function calls by listening for callmessages on an inport. If the process listens along multiple inports (using a *select* construct) than it can respond to multiple types of calls. Such a process can behave like an object, in that it supports multiple function calls (methods) that access its local data (instance variables). However it has much more flexibility: for example, it can choose to delay returning one callmessage until it receives another callmessage on a separate port. An example would be a barrier synchronization process that waited until it had received callmessage on each of a list of inports, and only then returned them all.

The type of a callmessage defines its parameters (in a syntax similar to a record definition), which parameters are constant, the typestate of the parameters on entry and return (which includes which parameters are initialized and which ones are not), and the exceptions that the callmessage can raise (and the typestates of the parameters for each exception). In total, a callmessage type defines a function call interface, as function signature does in Modula-3.

3.7 Procedures via Generator Processes

Processes and ports are asynchronous constructs. Callmessages are intended to provide standard procedure semantics on top of processes (although callmessages can also be used in other situations where you want to create an uncopyable token such that you'll be notified when it is destroyed). Another language construct, `procedure of` *code*, is a purely a syntactic convenience used in conjunction with `call` (which creates callmessages) and `return` (which returns callmessages) to provide standard procedure syntax as well. Where `create of` *code* just instantiates a process that execute code, `procedure of` *code* creates a generator process that keeps a copy of *code* as local data. Each time this process receives a callmessage it creates a *new* process that executes *code*, and passes the callmessage on to this new process.

This combination of callmessages, ports, processes, `call`, `return`, and `procedure of`, work together to implement normal procedure syntax and semantics in terms of the Hermes concepts. In the following example, the variable `multiply` stores code that stores into its first parameter the result of multiplying the values of its second and third parameter. Take the statements:

```
m <- procedure of multiply;
call m(x,17,23);
```

The first statement creates a process that creates multipliers and stores an outport for that process in `m`. The second statement creates a callmessage bundling `(x,17,23)` and sends that on `m`. At this point the process executing `call` goes to sleep. The process created in the first statment is listening on `m`'s corresponding inport, and when it gets the callmessage wakes up just long enough to create

a multiplier process and pass on the callmessage. This newly created multiplier process then does the multiplication, modifies the first field of the callmessage to contain the result, and sends it back with `return`. The callmessage returns immediately to the process that executed `call`, waking it. The callmessage is exploded into its components, at which point x receives its new value (this is value-result parameter passing). Although the underlying semantics are unusual, the net result is exactly the same as for function calls in languages that assume execution on a stack. Hermes adds one more bit of syntactic support by allowing properly defined outports to be used as functions as well as procedures, so that above example could be written completely equivalently as:

```
m <- procedure of multiply;
x <- m(17,23);
```

Because Hermes builds function calls out of more basic operations, it exposes more functionality. The programmer can use this functionality to implement different styles of function calls. The `procedure of` operation doesn't do anything that a normal Hermes program can't do, so a programmer could theoretically write his own process that accepted code and returned outports just as `procedure of` does (with the only caveat that this process would have to return these outports *wrapped* as *polymorphs* to satisfy the typechecker). In general the programmer can use `create of`, and `connect` directly to setup up any sort of communication paradigm he likes. For example, he can create a function that retains state across calls (actually these are simpler than functions that are created anew for each call). As another example, he can create a process that returns an aggregate of inports and then listens for calls on corresponding outports. If these inports act like a method suite than this process acts like an object. A process isn't constrained to use callmessages at all, and can instead communicate with its creator asynchronously: in this approach the creator sends his child at least one inport and one outport, and they communicate by explicit use of `send` and `receive`. The programmer can further combine or modify any of these communication styles or invent new ones.

Another place where the programmer might want to define his own `procedure of` function would be to provide better support for recursive function calls. Remember, a process must be explicitly passed outports for any function it calls, *including* itself. Therefore recursive functions have to be passed an outport connected to a generator process that can create more of them. For example a recursive factorial created with `procedure of` would have to be used as

```
f <- procedure of factorial;
x <- f(f,10);
```

Where the generator process returned by `procedure of` is used to create a new instantiation of the factorial process for each level of recursion. This is especially bad because if the programmer accidentally writes `create of` they will produce a single threaded factorial process that will deadlock when it tries to recurse! The generator process created by `procedure of` already forwards the callmessage; a user defined generator process designed for recursion could automatically include a recursion outport when it forwarded the callmessage.

3.8 Typestate

Typestate is an extension of the concept of type. Hermes uses it to provide static checks of many things that other languages either check dynamically or don't check at all: that you never reference through nil, use an unitialized variable, or misuse a variant. Modula-3 doesn't deal as elegantly with these concerns. Modula-3 deals with nil by inserting run-time checks before every dereference (at least semanticaly—in practice virtual memory can be used to catch these for free). Modula-3 avoids the problem of unitialized variables by initializing every variable to some meaningless but theoretically valid value (which causes a lot of variables to get "initialized" twice, once automatically by the language, and again then by the programmer). Modula-3 avoids the problem with variants by not having variants. Typestate also adds more information to interfaces, just as types do. This allows the compiler use local analysis to insure proper behaviour that would otherwise take global analysis to prove. Finally, just as type checking fails on some programs that actually behave properly,

so does typestate checking. The unproven assumption underlying Hermes typestate checking is that in these cases there is a better, safe way to write the same program. This assumption is plausible because Hermes includes ways to introduce dynamic typestate checking (such as polymorphs) that allow the programmer to do use unsafe operations at the cost of run-time checking.

A typestate is a set of attributes that a value of a given type may have. The most important typestate attribute is whether the memory for storing the value has been allocated. The attribute X is *init* is written $init(X)$. For scalar values $init(X)$ corresponds to whether X has been initialized (assigned a value) yet. The attribute *init* can also be applied to substructures of an aggregate, and to subsubstructures, and so on. A variant can store any of a set of types. Variants therefore have, as a typestate attribute, which type is actually being stored. The attribute V is in *case* $V.X$ is written $case(V, V.X)$.

For a program to type check (in the absense of subtypes) the compiler must be able to define a type for every variable (including expression temporaries) based on definitions, interfaces, and inference. Further, the operations on each variable must be must be consistent with its type. For a program to typestate check the compiler must be able to define a typestate for every variable at every program point such that its typestates of the variables are consistent with the operations that take place on them. For example, a variable must be *init* before it is accessed, and a variant V must be in the $case(V, V.X)$ to be accessed as $V.X$. The rules the compiler uses for typestate analysis are conservative: if there is any chance that an attribute is lost, than it is lost. Take the code example:

```
x <- 1;
if (b) then discard x; end if;
y <- x;
```

this example cannot typestate check. The possibility that `x` is discarded on the second line (and hence loses the attribute *init*) implies that `x` must be assumed to be unitialized on line 3, so the operation `y <- x` must be invalid. It is a good thing that this program doesn't typestate check, since if it was executed with value of true for `b` then it would cause a reference to deallocated memory.

Every Hermes operation also has preconditions and postconditions on the typestate of its arguments. Operations referencing a variables value typically have a precondition that it be *init*. Conversely, the assignment operation has a precondition that the destination variable not be *init*. Also there are implied conditions at the entry and exit of a process that all variables are unitialized. Interfaces (i.e. callmessage type definitions) must define the typestates of all parameters on entry and on exit. These form the pre and post conditions of any call to that that interface.

As stated above, whenever there is any possibility that a variable has lost an attribute, typestate analysis assumes it has. This can be specified precisely by saying that whenever two control flows join (such as at the end of an `if`) the typestate after the join is the least common denomiator (i.e. intersection) of the typestates before the join. This includes joins caused by looping constructs. The compiler will also automatically decrease the typestate (remove typestate attributes) of a variable to enable it to meet the preconditions of an operation. It does so by automatically inserting operations called *coercions* to lower the typestate. For attributes other than *init* this is just a matter of book-keeping. For example, the coercion *hide* is inserted to remove the attribute $case(V, V.X)$. This doesn't actually produce any code, it just makes any access of $V.X$ illegal (until this typestate attribute is restored). A dataflow algorithm similar to type resolution is used to determine typestates and where to insert coercions.

While most coercions are purely formal, the coercion corresponding to *init*, *discard*, actually deallocates storage. This is how Hermes solves the problem of safe storage allocation/deallocation, which Modula-3 solves with a garbage collector. This works because Hermes has no aliasing, so as soon as you know that the value in a variable is dead, you know you can deallocate the associated storage. Typestate analysis tells you exactly when values become dead. In Hermes all values are discardable, including communications types: inports, outports, and callmessages. Discarding an inport makes all the connected outports unusable. Discarding the last outport connected to an inport makes the inport unusable. Using a disconnected port produces an exception. Discarding of a callmessage automatically returns the callmessage to its creator with a depletion exception.

Just as a good static type checker insures that you don't have any run-time type errors, typestate checking insures that you don't have any of the run-time errors associated with unexpected values of typestate attributes.

In addition to *init* and *case*, the language defines two more attributes. The first is *checked*, defined for values of type `predefined!program` (which represents code). This attribute indicates the code has been checked for correctness and is ready to be instantiated as a process. *checked* is asserted as a precondition for the builtin operations `create of` and `procedure of`. Generally all interfaces that pass code mark it as *checked*. Code is only modified in a few routines that actually create new programs, and they are the only ones where checking occurs (in order to satisfy the typechecker a programmer must check code before he can return it through an interface that asserts that the code is *checked* on exit). There is a similar attribute for *checkeddefinitions* for definitions.

Users can also define constraint attributes, by giving a predicate (a pure boolean function) that determines when the constraint is met. For example, a user could define an attribute *positive* for integers (this is from page 118 of the Hermes manual) as:

```
positive: constraint (a:integer) is
{init(a)} a > 0;
```

For user defined constraints the language assumes that any modification of a value invalidates them, but that as long as the value is unchanged all contraints continue to be met (because constraints are defined by *pure* boolean functions).

User defined contraints is one of the nicest features of Hermes. By putting such constraints into interface definitions, the programmer can safely assume properties of the values he uses without checking them. He needs only insert checks where there is a possibility a value will be modified and then used in a context that requires a constraint. He then inserts a check with the statement `assert` *constraint(value)*. The compiler automatically inlines the correct predicate based on the definition of *constraint*. If the check fails, it raises an exception that can be caught and dealt with.

3.9 Exceptions

Exception handling is fundamental in Hermes: most basic operations in the language can return exceptions, such as *depletion*, which corresponds to a exceeding resource limitation such as memory or maximum integer size. User defined exceptions can also be defined and raised, and behave the same as the predefined exceptions. In Hermes an exception *never* "brings down the whole machine". Exceptions are always raised in the context of a process, just like they are raised in the context of a thread in Modula-3. In the case of exceptions caused by an operation, the exception is raised in the same process that executed the operation. Exceptions can also be raised while returning a callmessage: these exceptions occur in the process to which the callmessage is returned as it is reawakened. Within a process, dynamic scope and static scope are the same thing, so one can simply say that the exception jumps to the nearest enclosing handler. If there is no handler, the exception jumps to the end of the process (and then the process terminates).

The real cleverness is in how exception handling interacts with typestate checking. Basically, each only works because of the other. First off, typestate would be useless if it were not for exceptions, because any time an operation (including a function call) could possible fail to produce output in the proper typestate, the typestate checker would have to assume (conservatively) that it always fails. But there is always a chance that a procedure call will fail: for example, if the process trying to statisfy the procedure call runs out of memory. The way around this is that function (or actually callmessages) are allowed to return with exceptions. In addition to the typestates for a normal return, an interface defines the typestates for any exceptional return. The typestate checker incorporates this into the flow control graph it uses for its analysis. Because the scopes within a process are static, the typestate checker knows which handler will handle any exception raised at any point in the program. This means that the typestate checker can assume that after a function call the parameters are in the defined typestate, because the only way a function can return without meeting its normal typestate postconditions is to return with an exception, and in that case flow will jump instead to the handler.

Not only does exception handling make typestate usable, typestate makes exception handling work as well. Just as the semantics of callmessages make Hermes inter-process function calls act similarly to traditional function calls within a thread, the semantics of callmessages under *discard* combines with typestate checking to make exceptions propagate as they do traditionally. When a process fails to handle an exception the default "handler" is running off the end of the process code (and terminating the process). At the end of a process all variables (including the one holding the callmessage) must be uninitialized. This is enforced by the typestate checker by inserting discards, as necessary, along each flow control path leading to the end of the program. *This includes the flow control path caused by an exception.* The net result is that, as a side effect of typestate checking, the compiler automatically creates a default handler for any uncaught exceptions. This handler discards all initialzed variables and then exits the process.

The crucial twist is that the semantics of callmessages under discard don't deallocate the callmessages storage, but instead return the callmessage to the caller with a *depletion* exception. This has the effect of propagating uncaught exceptions up to the caller.

3.10 Data Types

Just like not everything Modula-3 is an object or an abstract data type, not everything in Hermes is a process. The language provides a an underlying set of data types and operations upon them. First the language provides the following scalar types.

- boolean
- integer
- real
- nominal

In order to maintain machine independence, the type *real* is parameterized by desired accuracy, expressed in terms of a fraction. While their intentions here are admirable, their approach is lousy: their definition of "accuracy" only makes sense for fixed point representations, while every major computer of the last 3 decades uses floating point. Not surprisingly, specifying accuracy is currently unsupported.

nominal is a special type of scalar value that represents a unique value, currently a `<machine,time>` stamp. Nominals are unforgable, and can be used as an alternative to ports for proving capabilities. A major use of nominals, however, is in place of pointers. One does this by defining a *table* of some type keyed by nominals. In this programming style the table acts like an address space and nominals as pointers, and table lookup corresponds to dereferencing a pointer. This is similar to the approach taken in languages like Fortran where, given that your data is stored in an array, integers are used to represent pointers to the data they index. As an example, a graph (as in mathematics) could be represented as a table of nodes keyed by nominals, where edges are represented as a separate table of pairs of nominals. Using nominals as pointers allows you to implement circular or recursive data structures in **Hermes**.

Hermes also defines the following agregate types:

- record
- table
- variant
- polymorph

All of these types except *polymorph* are type constructors: they define aggregate types in terms of other types. As expected, a *record* type consists of fields, each of whose type is defined.

A *table* type is defined in terms of the type (and typestate) of its elements and whether it is ordered or not. A table can also be keyed, which means that some component of its element type uniquely specifies the type. This is best seen in an example

```
employee: record (
name: string,
salary: real,
id: integer );
employee_table: table of element { full } keys (id);
```

This defines `employee_table` to be a type that can only be used to store `employees` all of which have distinct integers for field `id`. Any attempt to insert two employees with the same `id` will be checked and caught. The programmer can now also index an `employee_table` by `id` to retrieve the employee's entire record including their name. The decision to provide highly abstract tables with lots of functionality is based on two beliefs: first that they make programmers job easier, and second that a good compiler can choose an implementation of tables (such as arrays, hash tables, or binary trees) that reasonably efficient.

The types *variant* and *polymorph* both act to hide data. Variants can hold data in any of a finite set of types and typestates (specified in the variant's definition), while polymorphs can hold anything. They both act to introduce run-time checks to allow you to write programs that can not be statically checked. This is especially important because of Hermes typestatechecking. Modula-3 does the same thing with its type hierarchy: it introduces run-time checks whenever you try to assign a supertype to a variable that expects to hold a subtype.

Finally hermes supports the follow types for communication.

- callmessage
- inport
- outport

These were described earlier. Inports and outports are defined to transfer data of a specific type and typestate. You may only connect ports whose definition match.

3.11 Lazy Copy

Because there Hermes has no aliasing large structures must be copied to be shared. Rather than implement this naively, Hermes uses lazy copying. When the language semantics specify a copy, the implementation often just copies the pointer, resulting in two values that alias the same large structure. A reference count is kept in the structure of how many lazy copies have been made. In order to maintain proper semantics, any operation that modifies a large structure first checks the reference count. If there are no other copies the operation proceeds normally. If there are, however, the operation first produces a true copy of the structure, and uses this true copy instead. The reference counter on the lazy copied structure is decreased. A discard on a lazy reference also just decreases the counter, with the last holder of a reference (the one who decreases the counter to zero) finally deallocating it. This is purely an implementation issue: because Hermes provides an abstract machine model that hides these implementation details, the lazy implementation of copy preserves the same behaviour as the straightforward implementation.

4 Comparison by Features

4.1 Basic Programming Model

Both languages use a Algol style procedural programming. A thread of control executes a sequence of statements that refer to and modify variables. Both are structured languages, containing standard control structures for conditionals and loops. Code is organized into nested blocks which define scopes and can declare new variables. Both support traditional function calls and exception handling. Modula-3 has additional traditional features as well, while Hermes has additional nontraditional features.

4.1.1 Functions

Modula-3 and Hermes use traditional syntax for functional calls, and function calls work as expected. However, while **Modula-3** bases its functions on a traditional stack semantics, **Hermes** functions are merely one application of more general process semantics. You don't notice this when you make a normal function call, such as:

```
y := sin(x);
```

which works predictably in both languages. However you do notice this in areas like defining functions, passing functions as arguments, the scope of functions, and recursion.

In both Modula-3 and Hermes references to functions can be used as values. Modula-3 puts restrictions to avoid the problems of stack lifetimes that Pascal has. These problems only occur when pointers to nested procedures (procedures defined inside procedures) escape the lifetime of the enclosing procedures. Modula-3 avoids this by specifying that only references to top-level procedures can be assigned to procedure variables. This problem cannot occur when procedures are only passed down the stack, so Modula-3 has no restrictions on when procedures can be passed as arguments. Hermes doesn't need such restrictions because a process doesn't have an external environment or enclosing scope.

Hermes defines the compiler as part of the run-time environment, and so allows functions to be created dynamically.

4.2 Analyzability

Both languages carefully avoid holes that would lead to unpredictable behaviour. Because of this, both are much easier to analyze than insecure languages such as C. Hermes, however, goes much farther to support analyzability. In Hermes the behaviour of a process can be completely predicted by its internals, where in Modula-3 different procedures or threads can affect each others unpredictably through shared data. In Hermes there is no sharing of data between functions, and all operations on data are serialized to the thread of the process that owns the data. Herme's exception handling even insures predictability in the face of catastrophic failures. While Modula-3's exception handling and synchronization constraints allow the programmer to write correct code, they don't force the programmer to do so. For example, in Modula-3 you can't say for sure that a value on the heap hasn't changed between two instructions—there could always be another thread out there modifying it.

Herme's analyzability comes at the cost of a less traditional programming model. Programmers are used to shared memory. The assumption behind Hermes is that its analyzable programming model leads to fewer errors and is therefore worth learning.

Hermes analyzability theoretically would let it be compiled more efficiently than Modula-3. You need fewer dynamic checks because Hermes typestate encodes which values are nil. Hermes has no aliasing, so it can be compiled more efficiently on modern load-store architectures (once you've loaded a value into a register, you don't have to worry that a store to memory will invalidate that register). Lack of aliasing also makes the dependency graph of computations more predicable, with less false dependencies. This exposes more instruction level parallelism for better scheduling on superpipelined, superscalar architectures. Hermes also has advantages for parallelization: you can put each process on a different processor. Further, by combining computation with data, Hermes solves one of the biggest problems in current parallelizing compilers: how do you figure out which data to put on which processor. In practice, however, Modula-3 currently has good compilers while Hermes has only a mediocre interpreter. The same algorithm will run orders of magnitude faster in Modula-3 than in Hermes.

4.3 Modularity

Modula-3 supports multiple degrees of modularity. First off, all procedures are defined in the context of a module, with their interfaces explicitly defined and expressed in a separate file. On top of this, the information hiding mechanism allows the programmer to define abstract types (with a single

implementation), and the object oriented mechanism allows the programmer to define even more abstract entities (with multiple subclasses/implementations).

Herme's process semantics enforce module boundaries much stronger than anything in Modula-3. However, because all Hermes interfaces are defined in terms of exposed passive data types, Hermes interfaces often hide less. Where you would use an abstract data type or an object in Modula-3, in Hermes you have to choose between either exposing your data's structure or encapsulating it totally in a process. The later has a cost in efficiency (although in theory this is avoidable with better compilers). Worse, the syntax for manipulating data stored in a process is clumsier than for passive data. The programmer ends up packaging the methods that define an object himself.

For example, one might represent an abstract terminal in Hermes as a record containing two functions, one that returned the next character and one that accepted characters. This hides the implementation in the process that services the function calls, and allows the programmer to have multiple interchangable implementations (serial port terminals, X-window terminals, TCP/IP terminals) around at the same time. This is better than the standard trick of storing function pointers in a record because the Hermes functions are actually bound to a specialized process. To contrast, one would print the character 'X' on a Hermes terminal with `call term_var.out('X');`, where in C one might have the construct `term_var.out(&term_var,'X')` where the function has to be explicitly passed "self". In Hermes the programmer must write the extra code to bundle up methods in records.

As another example, where Modula-3 subtyping allows objects to be used as their supertype, Hermes doesn't. In Hermes you can again get the same functionality, this time by constructing multiple views of the same object. To continue the previous example, pretend you wanted a terminal with font support. In Modula-3 you could define a subclass FontTerminal to the class Terminal, and still use FontTerminal is routines that want a Terminal. In Hermes you would define and instantiate a process that provides the old functionality plus the new font support. You would store these functions in a record. When you wanted to use this record in the place of an old terminal, you would have to explicitly construct another record, copying into it just the functions appropriate for the old terminals, and pass that instead.

While these examples show that traditional object oriented programming in Hermes can be clumsy, Hermes also has advantages as well. For example, this separation of interface from implementation can be used to have multiple views of the same data. As an example, a graphics display can service functions that allow it to be viewed as a dumb terminal or an array of pixels, as well as functions for drawing 2D graphics and functions for drawing 3D graphics. The implementations of all of these are kept together where they share common data and private functions. One can combine these functions into method suites in different ways and so generate a whole set of "objects", all of which are different interfaces to the same underlying process.

4.4 Incremental Programming

Modula-3's object oriented inheritance has the advantage that minor changes are easy. Hermes supports a more flexible approach, closer to prototype based languages. One the standard ways to get incremental behaviour in hermes is to have a process filter messages for another. As in Self, the filtering process has no special access to its "supertypes" data, but instead uses the same interface everyone else does. This could be especially useful where you want to change an interface but still support the old one. Alternately, this can be achieved by having the process export multiple interfaces, as described above.

Modula-3 supports a different aspect of evolving interfaces with an `OBSOLETE` directive that allows a function to be kept in an interface, but generates a warning when it linked to. This provides a migration path. Once you've reorganized your code so that you don't generate any more warnings, you can trule remove all the obsolete functions.

4.5 Protection

Hermes provides richer and stronger protection than Modula-3. Hermes uses a capabilities model: what functions a process can call depends on what outports it has. This is complemented by unforgable objects, such as nominals. Each process has complete control over the inports on which it services function calls. If the process chooses to destroy (or not listen on) an inport, it effectively revokes that capability from everyone holding a connected outport. This approach, from operating systems, allows arbitrary protection schemes to be implemented in the same address space. Hermes also has advantages in terms of hiding: you can't tell anything about who is on the other end of an outport, except from their reaction to the messages you send.

Modula-3 has more limited protection and hiding. Modula-3's access control is based on who has the right to import what module. This must be defined statically by the outside system. Permissions tend to be all or nothing: if you can import an interface, you can call all the functions and use all the types it defines.

4.6 Error Handling

Hermes again has much more powerful error handling. A Hermes program can be written to survive any sort of failure. Processes act as firewalls, with the failure of one process only being reflected in the processes that later try to communicate with it. Proper language semantics are maintained in the face of all errors. While Modula-3 allows some support to help programs handle errors, these are only the errors detected by the program, not errors in the underlying system. The default behaviour for an error in Modula-3 is to bring down the whole program.

4.7 Concurrency

If you are going to have lots of concurrency, you are better off in Hermes. Hermes automatically synchronizes all data access by forcing each datum to be associated with a thread (process). Hermes programming style typically exposes more concurrency in your program. This is good in terms of efficiency on parallel machines, but bad in terms of reliability. It is much easier to write a program that deadlocks in Hermes than in Modula-3, where programmers are less likely to use concurrency casually.

Modula-3 does have good support for concurrency, by normal standards. It has a builtin concept of threads and synchronization primitives. It is less clear whether Modula-3 has problems with basic language constructs being safe in the face of concurrency. This is an implementation rather than a language definition issue. As a specific example, Modula-3 checks against references through nil. The question is what happens if another thread sets a value to nil in between when the check occurs and the reference occurs?

4.8 Portability

Both have a fairly abstract model of the machine that should promote portability. Hermes has minor advantages of greater abstraction that can lead to more efficient implementations. For example, Hermes tables might be implemented as a packed array on one machine and distributed arrays on another, and as an associative lookup table on an exotic machine with content-addressable memory. Hermes also runs naturally on either distributed or shared memory parallel systems, while Modula-3 is biased towards shared memory (which is going out of fashion). Hermes also tries harder to make scalar types like reals be machine independent. Modula-3 on the other hand allows some low-level operations that are specifically machine dependent, like specifying how many bits to store a number in.

4.9 Syntax

Where possible, both languages stick to algol style syntax. Hermes syntax is less compact than Modula-3. For example, you end up writing far too many type definitions. Just defining a procedure

forces you to define a callmessage type, and inport type, and an outport type. Also, doing object oriented programming in Hermes requires that the programmer do a lot of the book-keeping that Modula-3 does automatically. However, as a consequence, Hermes has a more flexible object oriented style.

5 Conclusion

There are tradeoffs between Hermes and Modula-3. Hermes makes it difficult to code synchronization errors, but easy to code deadlocks. Hermes gives the programmer complete control over the environment each process sees, but forces the programmer to implement his own linking protocol (such as passing around a huge record containing all library functions, or passing around a port to a library server). Hermes has a capabilities philosophy: if you don't want the programmer to do something, you don't allow him to do it. Modula-3 tries to make everything possible, but checks for mistakes. Where Modula-3 checks for references through nil, Hermes makes it impossible to legally reference a nil variable. Where Modula-3 provides functions for synchronization, Hermes only allows one thread access to a variable at a time. Overall, Modula-3 is a much more balanced, conservative language, where Hermes is radical. This may be why Modula-3 is being used while Hermes is not.

6 Bibliography

1. Robert E. Strom, et. al. "Hermes" *Prentice Hall*, 1991
2. Robert E. Strom, et. al. "Hermes Language Reference Manual" unpublished, 1987
3. Samuel P. Harbison, "Modula-3" *Prentice Hall*, 1992
4. Luca Cardelli, et. al., "Modula-3 Language Definition" *ACM Sigplan Notices*, volume 27, no. 8, August 1992