Variables and Parameters as References and Containers

Lawrence A. Crowl

Computer Science Department
Oregon State University
Corvallis, Oregon 97331-3202

Technical Report 92-60-20

November 1992

Abstract

Most designers of object-based languages adopt a reference model of variables without explicit justification, despite its wide ranging consequences. This paper argues that the traditional container model of variables is more efficient than the reference model, nearly as flexible, and more appropriate to parallel and distributed systems. The topics addressed are object lifetime and its implications for storage management, dynamic typing and its implications for object representation, aliasing and its implications for interference between operations, parameter passing and its implications for communication, and sharing and its implications for contention. We present our experience with the container model in a prototype parallel language. Neither model is always better than the other, and the choice of model should not be left to default.

Computing Reviews Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications — *concurrent, distributed and parallel languages; object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features — *concurrent programming structures; data types and structures; dynamic storage management; procedures, functions, and subroutines*

Keywords: object-based programming languages, reference variables, container variables, reference parameters, container parameters, variable lifetime, object lifetime, dynamic typing, static typing, dynamic allocation, static allocation, garbage collection, variable aliasing, parameter passing, communication, sharing, contention, parallelism, concurrency, distribution, Matroshka, Natasha

1 Introduction

A model of variables defines the nature of the information associated with a variable, and the manner in which that information may change. Since variables are a fundamental part of any programming language, the choice of a variable model affects the fundamental character of a programming language. In most procedure-based languages (*e.g.* Fortran and Algol) variables *contain copies* of values. We call this the *container* model of variables. In contrast, in most object-based languages (*e.g.* Smalltalk and Actors) variables *refer* to other objects. We call this the *reference* model of variables. (§1.1–1.3 elaborate these definitions.)

Programmers generally associate the container model with statically typed languages and fast execution, and the reference model with dynamically typed languages and flexible programming. The reference model is so pervasive in the object community that language designers often fail to mention their justification for adopting the model. In our view, the choice between these two models of variables is not so simple and deserves more explicit treatment by language designers. In particular, we argue that the container model will always be more efficient than the reference model, can be nearly as expressive, and is more appropriate to parallel and distributed systems. The topics we address are object lifetime and its implications for storage management (§2), dynamic typing and its implications for object representation (§3), aliasing and its implications for interference between operations (§4), parameter passing and its implications for communication (§5), and sharing and its implications for contention (§6). With our conclusions (§7), we present our experience with the container model in a prototype parallel language. We are not saying that designers of object-based languages should always choose the container model over the reference model, just that they should carefully consider the choice.

1.1 The Reference Model

In the reference model of variables, a variable refers to (points to) an object. (Variables may also be nil, referring to no object.) For example, figure 1 shows the relationship between objects and variables in the reference model. Two distinct variables may refer to the same object. In figure 1,

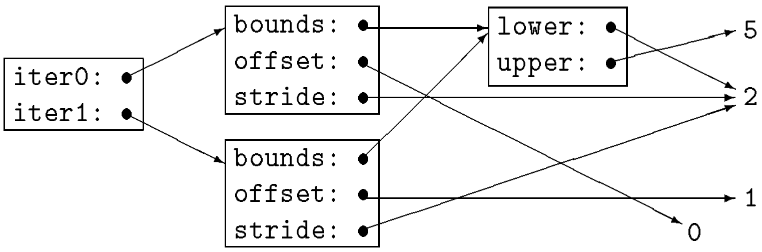

Figure 1: Reference Model of Variables

the two `bounds` variables refer to the same object, so changes to the `lower` variable is visible from the objects referred to by both `iter0` and `iter1`.

Since variables may only hold references, for consistency parameters and results of expressions must also refer to objects. Parameters and results by themselves will not change the state of an objects. In order for objects to change state, they must change (assign to) one of their variables. Assignment under the reference model consists of making a variable refer to an object. In our examples, we use the notation *variable* `<-` *expression* to indicate the binding of a variable to the result of an expression. We also use *variable*`.`*operation*`(`*arguments*`)` to indicate the invocation of an operation on an object. (Expressions bind left-to-right, as in C++.) For example, in the expression `x <- y.add(z)`, `y` and `z` are variables referring to integers, the result of the addition is a reference to another integer, and `x` is made to refer to that integer. Note that assignment is always possible, regardless of the objects referenced. Assignment is *not* a data operation, and may be implemented independently of any objects.

The reference model avoids an "infinite regression" in its pattern of objects being composed of references to other objects by grounding references in primitive objects, such as the integer 3. Programming systems must recognize these objects and implement them as something other than a set of references.

Since variables are not themselves objects, they cannot be referenced externally to an object's methods, and any changes to a variable must be done within an object method. This has implications for objects providing data structuring services. In particular, we must ask arrays to change their elements. For example, the Pascal statement `a[i]:=b[i]` would translate to `a.put(i,b.get(i))` where `put` and `get` correspond to the Smalltalk operations. Fortunately, the need for `put` only extends to one level of array access because outer data structures may simply return references to inner structures. For example, the Pascal statement `a[i][j]:=b[i][j]` translates to

```
a.get(i).put(j,b.get(i).get(j))
```

The reference model model places emphasis on object identity, and hence is appropriate to problems where identity is important, such as simulations of railroad networks.

1.2 The Strict Container Model

In the container model of variables, a variable contains an object. (Variables may also be nil, containing no object.) Figure 2 illustrates the relationship between objects and variables in the container model. Two distinct variables cannot refer to the same storage, as in the two `bounds`

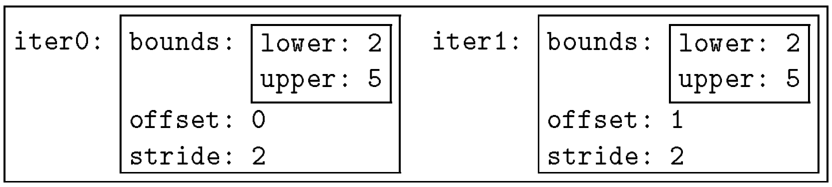

Figure 2: Container Model of Variables

variables in figure 2. Changing the `lower` variable in `iter0` will not affect the value in `iter1`.

Since variables hold entire objects, for consistency parameters and results of expressions must also be entire objects. When using a variable in an expression, we need access to its state, or value. Unlike in the reference model, objects in the container model must support a *copy* operation that permits us to obtain the current value of the object. The value is in essence an object unbound to any variable. Thus, given the variables `y`, we can obtain its value (another object with identical state) with the expression `y.copy()`, and use the object's value in an addition with `x.add(y.copy())`. This copy operation is implicit in most procedural programming languages via rvalue interpretation, with Bliss [Wulf *et al.*, 1975] a notable exception. In the remainder of this paper, we explicitly mark the copy operation, but due to its frequency, use the shorthand `#` in place of `.copy()`. The addition example becomes `x.add(y#)`. Note that objects for which a copy operation does not make sense (*e.g.* semaphores) need not support a copy operation.

Parameters and results by themselves will not change the state of an object. In order to change the state of an object, we must assign variables. Unlike in the reference model, there are two types of assignment in the container model:

binding: The binding assignment binds a variable to an object. This form is needed when changing the type of a variable. For example, in the expression `x <- y#`, the result of the copy is an object and `x` is bound to that object. (The expression `x <- y` is not legal because `y` is not a value.) Note that binding assignment is always possible, regardless of the objects referenced. Binding assignment is *not* a data operation, and may be implemented independently of the types of objects. The binding assignment is what gives the container model the flexibility to implement dynamic structures and types.

state-changing: Assignment may also mean a request to an object to change its state to match a given object. This form is useful when the type of the variables does not change. For example, in the expression `x.assign(y#)`, the result of the copy is an object as before, but it is then a parameter to the assignment operation on the object contained in the variable `x`. This means that assignment is a *data operation.* For those types which make sense, the type must provide a state-changing assignment operation. This form of assignment is common in procedural languages.

A programming language can usefully provide both forms of assignment. For example, in Natasha [Crowl, 1991] the definition of a variable includes a binding assignment, while subsequent assignments must use state-changing data operations.

In the container model, objects as containers grounds itself in primitive objects, recognized and implemented by the language processor.

Since references do not exist in a strict container model, data structuring objects cannot provide references to their components. This means, for example, that we must ask arrays to ask their elements to change state. The Pascal statement `a[i]:=b[i]` would translate to `a.put(i#,b.get(i#))`, which is the same as in the reference model except for the copy operation. Unlike the reference model, outer data structures cannot return references to inner structures and access to inner structures is more complex than in the reference model. For example, the Pascal statement `a[i][j]:=b[i][j]` might translate to

```
a.put(i#,a.get(i#).put(j#,b.get(i#).get(j#)))
```

which involves a substantial amount of data copying. An alternative is to make the outer structure aware of the type and operations of elements, and provide explicit operations to pass data through. The corresponding code might be

```
a.put2(i#,j#,b.get2(i#,j#)).
```

This awareness of element type and operations introduces undue complexity into the definition of data structuring objects. A better alternative is to have data structing objects provide an operation that applies another operation to its elements. With such an operation, the two-dimensional assignment becomes

```
a.atdo(i#,put(j#,(b.atdo(i#,get(j#)))))
```

In this example, `get(j#)` is not a function call, but is an operation name bound to a set of parameters, which when applied to an object will result in method invocation. With appropriate compiler support for in-lining operations, this operation passing approach can be as efficient as normal array access.

The container model places emphasis on object state, and hence is appropriate to problems where state is important but identity is not, such as physical simulations.

1.3 The Reference-Augmented Container Model

The reference model can *directly* represent an arbitrary graph of object relationships with its variables. On the other hand, the variables of the strict container model can only represent tree-structured (hierarchical) relationships. There are two solutions to this problem, the first is to require programmers to emulate references with indices into arrays, as they currently do in Fortran. The second solution is to introduce explicit *reference values*. With reference values, programmers may specify non-hierarchical relationships directly.

The presence of reference values permits data structuring objects to act much like they do in standard procedural languages. The Pascal statement `a[i]:=b[i]` would translate to

```
a.index(i#).assign(b.index(i#)#)
```

The use of references extends to multiple levels. In the two-dimensional example `a[i][j]:=b[i][j]`, the operations are

```
a.index(i#).index(j#).assign(b.index(i#).index(j#)#)
```

which is exactly the code that a Pascal compiler generates. Note that this sequence of operations does not require sophisticated compiler support to execute efficiently.

The reference-augmented container model is appropriate when state is most important, but identity may also be important. It may also be appropriate when compiler support is minimal.

1.4 The C++ Variable Model

Variables in C++ are clearly containers for the language's primitive types [Ellis and Stroustrup, 1990]. Since C++ also provides pointers, we could infer that C++ provides a reference-augmented container model. However, C++ facilities for dynamic typing and virtual functions only work when using pointers, so programmers must restrict themselves to pointers to all objects that need the full expressiveness of the language. Practically, C++ provides a reference model of variables for some objects and the reference-augmented container model for other objects.

2 Object Lifetime and Storage Management

Under the reference model, references to an object may spread freely. Because objects are often extensively shared, the programming system cannot reasonably delete objects until they are no longer referenced, and the lifetimes of objects cannot always be associated with the lifetimes of the variables that reference them. The indeterminate lifetime of objects implies dynamic heap allocation and asynchronous garbage collection of objects, which can be expensive. When compiler analysis can associate the lifetime of an object with the lifetime of a variable [Ruggeri and Murtagh, 1988], or with the lifetime of another object [Hutchinson, 1987; Hayes, 1991], the compiler can insert explicit deallocation operations and improve performance of storage management.

To reclaim storage (collect garbage), the system must examine the entire set of references in the system to insure that no references to an object exist before deleting the object. This is possible on multiprocessors [Appel *et al.*, 1988], but on large or widely distributed systems, the number of possible locations for a reference becomes very large, substantially affecting the methods and expense of garbage collection [Liskov and Ladin, 1986; Eckart and LeBlanc, 1987; Schelvis, 1989; Ladin and Liskov, 1992].

On the other hand, under the strict container model, objects may *not* be referenced when the variable containing them no longer exists — objects have a lifetime corresponding exactly to that of the variable containing them. This enables compilers to explicitly deallocate all objects, which makes deallocation synchronous and relatively efficient. Explicit deallocation makes space and time for storage management more predictable, which is important for real-time systems. In addition, whenever procedures are activated within a stack discipline, their variables' objects may be allocated on a stack, further increasing the efficiency of storage management. (We consider the profligate use of virtual address space [Appel, 1987] to have limited applicability.)

The augmented container model re-introduces the situation where references to an object may persist after the corresponding variable, which may result in dangling references. There are three solutions to this problem. First, one may define programs with dangling references as erroneous, as in C [Kernighan and Ritchie, 1988]. Second, one may restrict the propagation of references so that dangling references may not occur, as in Algol 68 [van Wijngaarden *et al.*, 1976]. Finally, one may define objects to exist as long as references to them exist, as in Ada [U. S. DoD, 1983]. In the last solution, we re-introduce the storage management problems of the reference model, but since there will likely be fewer references, compiler analysis should be more effective.

Storage management costs are no worse in the container model than in the reference model, and may be substantially better.

3 Dynamic Typing and Object Representation

The dynamic allocation of objects imposed by the unstructured lifetimes of object under the reference model permits dynamic typing with little additional storage. Each object need only contain a pointer to its type. Figure 3 illustrates this structure.

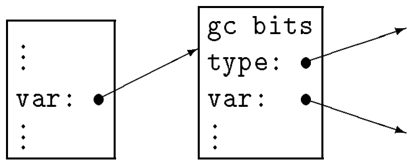

Figure 3: Variable and Object Structure in the Reference Model

Most reference-based languages define many primitive objects to be immutable, meaning that no operation will change their value. The implementation is free to in-line the representations of such objects in place of references to them [Goldberg and Robson, 1983; Black *et al.*, 1986a]. For example, instead of pointers to integers, implementations may use the integers themselves. In-lining is typically used only for pointer-sized objects. In a dynamically typed language, the pointers and integers must generally be tagged to distinguish between them, though compiler analysis will help eliminate some tags [Kaplan and Ullman, 1978; Mishra and Reddy, 1985; Boehm, 1989; Wand, 1989; Aiken and Murphy, 1991; Lincoln and Mitchell, 1992]. Figure 4 illustrates the in-lining of primitive objects and the associated tagging.

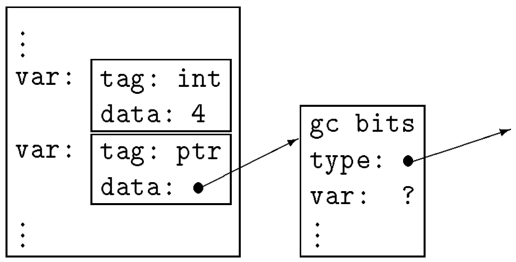

Figure 4: In-lining Primitive Objects in the Reference Model

Implementations of the container model can use the object representations described above for the reference model. Indeed, when the variable may not be bound, or may be bound to objects of different types[1], some form of indirection is required. The implicit indirection enables the container model to support dynamic allocation and typing, which in turn supports structures like variable-length polymorphic lists. However, the indirection is not visible to the programmer and the programmer has no way to determine that the indirection is actually present, and no way to create an alias to the object (§4). Thus, the container model can support dynamic typing in a manner similar to the reference model.

[1]We use type to indicate implementation, and not interface because it is the implementation which is relevant to representation.

In the presence of static typing, more efficient object representations are available. In particular, if the type of a variable is static (through language definition or compiler analysis) and there is always an object defined for that variable, the implementation may place the representation of an object inside the representation for another. Figure 5 illustrates this in-lining. We can remove

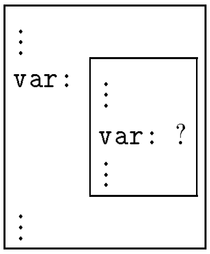

Figure 5: In-lining Staticly-Typed Objects in the Container Model

the type pointer from the definition of objects by making use of type-specific garbage collection routines [Goldberg, 1991; Diwan *et al.*, 1992]. In-lined representations also remove the storage and interpretation required for tags and pointers. This optimization is particularly important for arrays of simple objects. This optimization is possible for the reference model, but requires the compiler to demonstrate both static type and static lifetime.

The reference-augmented container model may introduce a requirement to reference an in-lined object from a context in which the type of the reference is not known staticly. This problem is easily solved by associating the type pointer with the data pointer rather than with the object itself. Figure 6 illustrates this approach. This representation will not use an undue amount of storage,

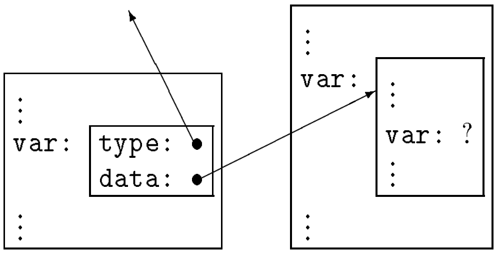

Figure 6: Dynamicly Typed References in the Reference-Augmented Container Model

provided that most small data items have static types and are in-lined. This representation can also serve when the references are used implicitly to support dynamic allocation and typing.

The container model can support dynamic allocation and typing as efficiently as the reference model, and can support static allocation and typing more efficiently than the reference model.

4 Aliasing and Operation Interference

In the reference model, the state of an object is determined by its set of references. Since the change in state of one object may change the behavior of objects that reference it, we also define the extended state of an object to be the transitive closure of the extended states of the objects it references. Because two objects may have variables referencing a third object, operations ostensibly on the first object may change the extended state of the second object. Improperly managing this aliasing will introduce obscure, unexpected and difficult-to-debug interference between operations. The reference model makes strict encapsulation difficult. This problem was deemed significant enough that Hogg [1991] introduced additional language mechanisms to control aliasing.

Also in the reference model, it is not generally possible to determine *a priori* when two variables will refer to distinct objects. The result is that the programmers and language systems must assume that the variables may refer to the same object (until proven otherwise). This potential aliasing inhibits dependency detection, which inhibits the detection and exploitation of parallelism by both programmers and compilers. Given that a potential alias exists, programmers and compilers must either avoid any attempt to operate on the two variables concurrently, or enforce mutual exclusion within each potentially aliased object.

In contrast, the strict container model ensures that each variable contains a different object, and that each object is contained within a single variable. The state of a variable is simply the state, or value, of the object it contains. The state of an object is the Cartesian product of the states of its variables. Since only the states of variables contained within an object can affect the behavior of the object, operations on one object cannot affect the state of another and we do not need a notion of the extended state of objects. Strict encapsulation is a by-product of the variable model. Thus errors due to aliasing are considerably reduced. In addition, the programmer and the compiler are free to operate on two different variables concurrently. We expect parallelizing programs based on the container model to be no more difficult than parallelizing equivalent Fortran programs, for which there is considerable expertise [Sarkar and Hennessy, 1986; Allen *et al.*, 1987; Polychronopoulos, 1988; Wolfe, 1989; Sarkar, 1990]. The current research effort in parallelizing sequential programs may aid in further parallelizing parallel programs by inferring unstructured parallelism within the structured parallelism that programmers typically provide.

The augmented container model re-introduces the difficulties of performing alias detection, but with fewer aliases than would occur in the reference model. Again, there is substantial research devoted to this problem [Cooper, 1985; Landi and Ryder, 1991; Landi and Ryder, 1992].

The container model can substantially reduce the problem of aliasing, and in any case will be no worse than the reference model.

5 Parameter Passing and Communication

One consequence of the reference model is that parameters are passed by reference. This parameter mechanism is called pass-by-sharing in CLU [Liskov *et al.*, 1977]. This parameter mechanism may cause unacceptable performance penalties on distributed systems due to heavy communication between machines as operations traverse back and forth across machine boundaries to reach the objects referenced by the parameters. For example, if we pass `iter0` of figure 1 as a parameter to an operation for another machine, the operation on that machine must then return to the original machine for the `bounds` variable and then again for the `lower` variable. This approach results in five messages between processors, which is quite high for such a small amount of information.

The problem of increased communication due to the reference model is severe enough that Argus [Liskov and Scheifler, 1983], which uses CLU's reference model on a single machine, uses a container model for parameters between machines [Herlihy and Liskov, 1982]. On the other hand, Emerald [Black *et al.*, 1986b] uses the reference model both within and between machines. Emerald mitigates the cost of remote arguments with three mechanisms, *call-by-move*, *call-by-visit*, and *attach*. Call-by-move indicates that the argument object is to be moved to the node containing the called object. Call-by-visit indicates that the argument object is to be moved to the node containing the called object for the duration of the call and then moved back to the caller's node upon completion. Attach indicates that one object should be moved whenever the object it is attached to moves. The Emerald compiler attempts to determine when an object is only referenced from within another and therefore can be implicitly attached to the other.

Note that even with object movement under the reference model, the objects are still potentially referenced and accessed from other machines and therefore are still possible sources of contention (§6).

In contrast, systems using the container model pass all data associated with a parameter at the same time as it passes the parameter. For example, passing `iter0` of figure 2 will also pass its `bounds`. This approach requires only one message between processors.

One possible objection to the container model is that the copying of parameter objects may introduce unacceptable overhead when processors share memory. We believe this problem is manageable because:

- For small objects, the most efficient way to pass parameters is in registers, which is copying.
- The object model reduces the need to pass large objects because the object operated on is not a parameter and is not copied. So for operations involving one small object and one large object, programmers can make the large object the target of the operation. For example, in table lookup, programmers would make the table the target of the operation, and the index would be the parameter.
- For non-uniform memory access multiprocessors supporting block copy, such as the BBN Butterfly, it is often more efficient to copy a large object across processors once rather than access individual words remotely, even when a substantial fraction of the words are not used.
- For large parameter objects, the reduced aliasing properties of the container model (§4) may permit the implementation to determine that the parameter may be safely aliased with the source object, and hence passed by reference.
- Finally, systems with virtual memory may copy large objects with copy-on-write.

The container model also permits a language to unify message passing and shared memory into operation invocation [Crowl, 1991]. A programming system is then free to implement operations on objects as either local procedures on local data, local procedures on remote data, and remote procedures on remote data. The ability to choose the appropriate implementation can substantially improve performance [Fowler and Kontothanassis, 1992; Cox *et al.*, 1992; LeBlanc and Markatos, 1992].

Where object identity is not important, the container model of variables and parameters permits the implementation to reduce the communication costs by sending fewer messages where messages are appropriate and by using remote procedure calls or shared memory where they are appropriate.

6 Sharing and Contention

Whenever parallel or distributed programs share information (objects), they introduce the possibility of contention for those objects. While some contention is unavoidable, excessive contention leads to poor performance. So, one should write parallel and distributed programs in a manner that shares objects as little possible, and therefore choose a model of variables that aids this task.

When sharing actively changing (mutable) objects, contention is inevitable and the choice of the model of variables will have little effect on contention. When sharing objects that never change (immutable objects), both the reference and container models may copy the objects freely so as to reduce contention.[2]

The two models of variables differ when sharing the current state of an object. Many objects may be accessed in phases, where for long periods of time the program is interested in an object's current state, and not in tracking its changes in state. For example, in LU-decomposition each row changes state until it becomes the pivot row, at which point further reductions need only the value of the row. To reduce contention in the reference model, the programmer has two options: to make the rows immutable or to copy the pivot row explicitly. Both approaches have drawbacks, however:

- When making the rows immutable, the compiler is free to copy each argument row to the local node. Unfortunately, this involves increased dynamic allocation and deallocation of row objects in the normal maintenance of the matrix because immutable rows cannot be updated in-place. The new value of the row must be computed in new storage. In the container model, programmers may pass copies of the row, rather than making them immutable. Each processor accesses a copy of the row, thus reducing contention.

- Because one cannot generally determine the lifetime of objects in the reference model, an explicit copy increases the amount of dynamic allocation and deallocation within the system. In contrast, the implicit copy under the container model has a fixed lifetime and can have more efficient allocation and deallocation. The explicit copy in the reference model is analogous to the implicit copy in the container model, but with higher run-time and programmer costs.

The reference model encourages the sharing of objects, which is inappropriate for parallel and distributed systems. In contrast, the container model discourages the sharing of objects which are not necessarily shared.

[2]Note that under some systems, such as Emerald [Hutchinson, 1987], the representation of an immutable object may change over time, so long as its abstract value does not. The change in representation enables objects to adapt to changes in access patterns.

7 Experience and Conclusions

Our programming experience with the container model includes five programs implementing significant parallel application kernels and several other small programs. These programs represent over 1400 lines of code in Natasha, a staticly-typed prototype language and implementation of the Matroshka parallel programming model [Crowl, 1991], which is based on the reference-augmented container model.

The association of object lifetime with variable lifetime in the container model enables more predictable and efficient storage management than the reference model. The container model can accommodate dynamic typing as efficiently as the reference model, and can make better use of storage when types are static. We have no direct experience with dynamic types in the container model because Natasha is statically typed, but we can attest to the effectiveness of in-lining staticly-typed objects.

The reference model naturally encourages aliasing, which violates encapsulation and discourages parallelism. In contrast, the container model enforces encapsulation and encourages parallelism. Because of the enforced encapsulation, the strict container model requires more compiler support to efficiently access data structures than do either the reference model or the reference-augmented container model. Natasha provides reference values, variables and parameters, but our example programs used reference values exclusively for access to elements of data structuring objects. The use of reference values occurred whenever we used arrays, but almost exclusively as intermediate values in expressions. We never declared reference variables and declared reference parameters only six times, and then in the definition of iterators for data structures. Had we anticipated a compiler with significant in-lining capabilities, we would have avoided reference parameters using the technique mentioned in §1.2. We believe that a strict container model will serve for many applications.

The copying of parameters implied by the container model improved the performance of parallel LU-decomposition on the BBN Butterfly, even when passing large vectors [Crowl, 1991]. The container model is more appropriate for parallel and distributed environments because it encourages communication through copied parameters, rather than through shared references.

The many advantages of the container model of variables and parameters over the reference model, both in semantics and implementation, particularly with respect to parallel and distributed programming, indicate that the container model deserves more explicit treatment by the object language community as well further research into its costs and benefits.

Acknowledgments

We thank Timothy Budd, Thomas LeBlanc and Rajeev Pandey for their comments.

References

[Aiken and Murphy, 1991] A. Aiken and B. Murphy, "Static Type Inference in a Dynamically Typed Language," In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 279–290, Orlando, FL, January 1991.

[Allen *et al.*, 1987] Randy Allen, David Callahan, and Ken Kennedy, "Automatic Decomposition of Scientific Programs for Parallel Execution," In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 63–76, January 1987.

[Appel *et al.*, 1988] A. W. Appel, J. R. Ellis, and K. Li, "Real-time Concurrent Collection on Stock Multiprocessors," In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20, Atlanta, Georgia, June 1988, appeared in ACM SIGPLAN Notices 23(7).

[Appel, 1987] Andrew W. Appel, "Garbage Collection Can Be Faster Than Stack Allocation," *Information Processing Letters*, 25(4):275–279, 1987.

[Black *et al.*, 1986a] Andrew P. Black, Norman Hutchinson, Eric Jul, and Henry Levy, "Object Structure in the Emerald System," In *Proceedings of 1986 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 78–86, September 1986, appeared in ACM SIGPLAN Notices 21(11), November 1986.

[Black *et al.*, 1986b] Andrew P. Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter, "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering*, December 1986.

[Boehm, 1989] Hans-Juergen Boehm, "Type Inference in the Presence of Type Abstraction," In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 192–206, Portland, Oregon, June 1989, also in ACM SIGPLAN Notices 24(7), July 1989.

[Cooper, 1985] Keith D. Cooper, "Analyzing Aliases of Reference Formal Parameters," In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 281–290, New Orleans, Louisiana, January 1985.

[Cox *et al.*, 1992] Alan L. Cox, Robert J. Fowler, and Jack E. Veenstra, "Interprocessor Invocation on a NUMA Multiprocessor," Technical Report 356, Computer Science Department, University of Rochester, September 1992.

[Crowl, 1991] Lawrence A. Crowl, "Architectural Adaptability in Parallel Programming," Technical Report 381, Computer Science Department, University of Rochester, May 1991, Ph.D. Dissertation.

[Diwan *et al.*, 1992] A. Diwan, E. Moss, and R. Hudson, "Compiler Support for Garbage Collection in a Statically Typed Language," In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992.

[Eckart and LeBlanc, 1987] J. D. Eckart and Richard J. LeBlanc, "Distributed Garbage Collection," In *Proceedings of the ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 264–273, St. Paul, Minnesota, June 1987, also in ACM SIGPLAN Notices 22(7), July 1987.

[Ellis and Stroustrup, 1990] Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, 1990.

[Fowler and Kontothanassis, 1992] Robert J. Fowler and Leonidas I. Kontothanassis, "Improving Processor and Cache Locality in Fine-Grain Parallel Computations Using Object-Affinity Scheduling and Continuation Passing," Technical Report 411 (revised), Computer Science Department, University of Rochester, June 1992.

[Goldberg and Robson, 1983] Adele Goldberg and David Robson, *Smalltalk-80, The Language and Its Implementation*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[Goldberg, 1991] B. Goldberg, "Tag-Free Garbage Collection for Strongly Typed Programming Languages," In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, June 1991, also in ACM SIGPLAN Notices 26(6), June 1991.

[Hayes, 1991] Barry Hayes, "Using Key Object Opportunism to Collect Old Objects," In *Proceedings of 1991 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 33–46, October 1991.

[Herlihy and Liskov, 1982] Maurice P. Herlihy and Barbara H. Liskov, "A Value Transmission Method for Abstract Data Types," *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[Hogg, 1991] John Hogg, "Islands: Aliasing Protection in Object-Oriented Languages," In *Proceedings of 1991 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 271–285, October 1991.

[Hutchinson, 1987] Norman C. Hutchinson, "Emerald: An Object-Based Language for Distributed Programming," Technical Report 87-01-01, Department of Computer Science, University of Washington, January 1987, Ph.D. Dissertation.

[Kaplan and Ullman, 1978] Marc A. Kaplan and Jeffery D. Ullman, "A General Scheme for the Automatic Inference of Variable Types," In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 60–75, January 1978.

[Kernighan and Ritchie, 1988] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall Inc., Englewood Cliffs, New Jersey 07632, second edition, 1988.

[Ladin and Liskov, 1992] Rivka Ladin and Barbara Liskov, "Garbage Collection of a Distributed Heap," In *Proceedings of the Twelfth ICDCS*, Yokohama, Japan, June 1992.

[Landi and Ryder, 1991] W. Landi and B. G. Ryder, "Pointer-Induced Aliasing: A Problem Taxonomy," In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, Orlando, Florida, January 1991.

[Landi and Ryder, 1992] W. Landi and B. G. Ryder, "A Safe Approximate Algorithm for Interprocedural Pointer Aliasing," In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, San Francisco, California, June 1992, also in ACM SIGPLAN Notices 27(7), July 1992.

[LeBlanc and Markatos, 1992] Thomas J. LeBlanc and Evangelos P. Markatos, "Shared Memory vs. Message Passing in Shared-Memory Multiprocessors," In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, December 1992.

[Lincoln and Mitchell, 1992] P. Lincoln and J. C. Mitchell, "Algorithmic Aspects of Type Inference with Subtypes," In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 293–304, Albuquerque, New Mexico, January 1992.

[Liskov and Ladin, 1986] Barbara Liskov and Rivka Ladin, "Highly Available Distributed Services and Fault-Tolerant Distributed Garbage Collection," In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1986. ACM.

[Liskov and Scheifler, 1983] Barbara H. Liskov and Robert Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[Liskov *et al.*, 1977] Barbara H. Liskov, Alan Snyder, R. R. Atkinson, and J. C. Schaffert, "Abstraction Mechanisms in CLU," *Communications of the ACM*, 20(8):564–576, August 1977.

[Mishra and Reddy, 1985] Prateek Mishra and Uday S. Reddy, "Declaration-Free Type Checking," In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 7–21, January 1985.

[Polychronopoulos, 1988] Constantine D. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988.

[Ruggeri and Murtagh, 1988] Cristina Ruggeri and Thomas P. Murtagh, "Lifetime Analysis of Dynamically Allocated Objects," In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[Sarkar, 1990] Vivek Sarkar, "PTRAN — The IBM Parallel Translation System," Technical Report RC-70566, Research Division, IBM T. J. Watson Research Center, Yorktown Heights, New York, May 1990.

[Sarkar and Hennessy, 1986] Vivek Sarkar and J. Hennessy, "Compile-time Partitioning and Scheduling of Parallel Programs," In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, pages 17–26, July 1986.

[Schelvis, 1989] Marcel Schelvis, "Incremental Distribution of Timestamp Packets: A New Approach to Distributed Garbage Collection," In *Proceedings of 1989 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 37–48, New Orleans, Louisiana, October 1989, also appeared in ACM SIGPLAN Notices 24(10), October 1989.

[U. S. DoD, 1983] United States Department of Defense, Washington D. C., *Reference Manual for the Ada Programming Language*, June 1983, ANSI/MIL-STD-1815A.

[van Wijngaarden *et al.*, 1976] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Linsey, L. G. L. T. Meertens, and R. G. Fisker, *Revised Report on the Algorithmic Language Algol 68*, Springer-Verlag, 1976.

[Wand, 1989] Mitchell Wand, "Type Inference for Record Concatenation and Multiple Inheritance," In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, 1989.

[Wolfe, 1989] Michael J. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, 1989.

[Wulf *et al.*, 1975] William A. Wulf, Richard K. Johnsson, Charles B. Weinstock, Steven O. Hobbs, and Charles M. Geschke, *The Design of an Optimizing Compiler*, Elsevier, 1975.