ParaSail: A Pointer-Free Path to Object-Oriented Parallel Programming

S. Tucker Taft

AdaCore
24 Muzzey Street
Lexington, MA 02421 USA
taft@adacore.com

Abstract

Pointers are ubiquitous in modern object-oriented programming languages, and many data structures such as trees, lists, graphs, hash tables, etc. depend on them heavily. Unfortunately, pointers can add significant complexity to programming. ParaSail, a new parallel object-oriented programming language, has adopted an alternative, pointer-free approach to defining data structures. Rather than using pointers, ParaSail supports flexible data structuring using *expandable* (and shrinkable) objects, along with generalized indexing. By eliminating pointers, ParaSail significantly reduces the complexity for the programmer, while also allowing ParaSail to provide pervasive, safe, object-oriented parallel programming.

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 – abstract data types, classes and objects, concurrent programming structures, polymorphism.

General Terms Algorithms, Design, Reliability, Languages, Theory, Verification.

Keywords pointer-free; region-based storage management; expandable objects; parallel programming.

1. Introduction

Pointers are ubiquitous in modern object-oriented programming languages, and many data structures such as trees, lists, graphs, hash tables, etc. depend on them heavily. Unfortunately, pointers can add significant complexity to programming. Pointers can make storage management more complex, pointers can make assignment and equality semantics more complex, pointers can increase the ways two different names (access paths) can designate the same object, pointers can make program analysis and proof more complex, and pointers can make it harder to *divide and conquer* a data structure for parallel processing.

Is there an alternative to using pointers? ParaSail[1], a new parallel object-oriented programming language, adopts a different paradigm for defining data structures. Rather than using pointers, ParaSail supports flexible data structuring using *expandable* (and shrinkable) objects, along with generalized indexing. By eliminating pointers, ParaSail significantly reduces the complexity for the programmer, while also allowing ParaSail to provide pervasive, safe, object-oriented parallel programming.

2. Expandable and Optional Objects

An *expandable* object is one that can grow without using pointers, much as a house can grow through additions. Where once there was a door to the back yard, a new screened-in porch can be added. Where once there was only one floor, a new floor can be added. The basic mechanism for expansion in ParaSail is that every type has one additional value, called **null**. A component can initially be null, and then be replaced by a non-null value, thereby expanding the enclosing object. At some later point the enclosing object could shrink, by replacing a non-null component with null.

Not every component of an object is allowed to be null. The component must be declared as **optional** if it is allowed to take on a null value. For example, a `Tree` structure might have a (non-optional) `Payload` component, and then two additional components, `Left` and `Right`, which are each declared as **`optional`** `Tree`. Similarly, a stand-alone object may be declared to be of a type `T`, or of a type **`optional`** `T`. Only if it is declared **optional** may it take on the null value. The value of an object X declared as **optional** may be tested for nullness using `X` **`is null`** or `X` **`not null`**.

Another example of a data structure using optional components would be a linked list, with each node having two components, one `Payload` component, and a `Tail` component of type **`optional`** `List`. There is also a built-in parameterized type, `Basic_Array<Component_Type>` which allows the Component_Type to be specified as **optional**. This allows, for example, the construction of a hash table with buckets represented as linked-lists, by declaring the *backbone* of the hash table as a `Basic_Array<`**`optional`** `List<Hash_Table_Item>>`. The components of the hash table would start out as **null**, but as items are added to the hash table, one or more of the component lists would begin to grow.

2.1 Assignment, Move, and Swap Operations

Because there are no pointers, the semantics of assignment in ParaSail are very straightforward, namely the entire right-hand-side object is copied and assigned into the left-hand side, replacing whatever prior value was there. However, there are times when it is desirable to *move* a component from one object to another, or *swap* two components. Because implementing these on top of an assignment that uses copying might impose undue overhead, in ParaSail, *move* and *swap* are separate operations. The semantics of *move* are that the value of the left-hand-side is replaced with the value of the right-hand-side, and the right-hand-side ends up null. For *swap*, the values of the left- and right-hand-side are swapped. Syntactically, ParaSail uses ":=" for (copying) assignment, "<==" for move, and "<=>" for swap. The ParaSail compiler is smart enough to automatically use *move* semantics when the right-hand-side is the result of a computation, rather than an object or component that persists after the assignment.

As an example of where *move* might be used, if our hash table grows to the point that it would be wise to lengthen the backbone, we could create a new `Basic_Array` twice as large (for example), and then *move* each list node from the old array into the new array in an appropriate spot, rebuilding each linked list, and then finally *move* the new array into the original hash-table object, replacing the old array. The *swap* operation is also useful in many contexts, for example when balancing a tree structure, or when sorting an array.

2.2 Binary Tree Example

The Appendix includes an example of a pointer-free tree-based map module implemented in ParaSail. This example illustrates the use of **optional** components, as well as "**<==**" (*move)*, which is used as part of a Delete operation.

3. Cyclic Data Structures and Generalized Indexing

Expandable objects allow the construction of many kinds of data structures, but a general, possibly cyclic graph is not one of them. For this, ParaSail provides generalized indexing. The array-indexing syntax, "`A[I]`," is generalized in ParaSail to be usable with any container-like data structure, where `A` is the container and `I` is the key into that data structure. A directed graph in ParaSail could be represented as a table of Nodes, where the index into the table is a unique node `Id` of some sort, with edges represented as Predecessors and Successors components of each Node, where Predecessors and Successors are each sets of node-ids. See the second example in the Appendix for an illustration of using generalized indexing to represent a directed graph.

If edges in a directed graph are represented with pointers, it is possible for there to be an edge that refers to a deleted node, that is, a *dangling* reference. Such a dangling reference could result in a storage leak, because the target node could not be reclaimed, or it could lead to a potentially destructive reference to reclaimed storage. When edges are represented using node-ids, there is still the possibility of an edge referring to a deleted node or the wrong node, but there is no possibility for there to be associated storage leakage or destructive reference to reclaimed storage, as node-ids are only meaningful as keys into the associated container.

4. Region-Based Storage Management

Storage management without pointers is significantly simplified. All of the objects declared in a given scope are associated with a storage *region*, essentially a local heap. As an object grows, all new storage for it is allocated out of this region. As an object shrinks, the old storage can be immediately released back to this region. When a scope is exited, the entire region is reclaimed. There is no need for asynchronous garbage collection, as garbage never accumulates.

Every object identifies its region, and in addition, when a function is called, the region in which the result object should be allocated is passed as an implicit parameter. This *target* region is determined by how the function result is used. If it is a temporary, then it will be allocated out of a temporary region associated with the point of call. If it is assigned into a longer-lived object, then the function will be directed to allocate the result object out of the region associated with this longer-lived object. The net effect is that there is no copying at the call site upon function return, since the result object is already sitting in the correct region.

Note that pointers are still used *behind the scenes* in the current implementation of ParaSail, but eliminating them from the surface syntax and semantics eliminates essentially all of the complexity associated with pointers. That is, a semantic model of expandable and shrinkable objects, operating under (mutable) *value* semantics, rather than a semantic model of nodes connected with pointers, operating under *reference* semantics, provides a number of benefits, such as simpler storage management, simpler assignment semantics, easier analyzability, etc.

The move and swap operations have well-defined semantics independent of the region-based storage management, but they provide significant added efficiency when the objects named on the left and right-hand side are associated with the same region, because then their dynamic semantics can be accomplished simply by manipulating pointers. In some cases the programmer knows when declaring an object that it is intended to be moved into or swapped with another existing object. In that case, ParaSail allows the programmer to give a *hint* to that effect by specifying in the object's declaration that it is "**`for`** `X`" meaning that it should be associated with the same region as X. With region-based storage management, it is always safe to associate an object with a longer-lived region, but to avoid a storage leak, the ParaSail compiler will set the value of such an object to null on scope exit, as its storage would not otherwise be reclaimed until the longer-lived region is reclaimed. An optimizing compiler could automatically choose to allocate a local variable out of an outer region when it determines that its last use is a move or an assignment to an object from an outer region.

5. Parallel and Distributed Programming

In addition to removing pointers, certain other simplifications are made in ParaSail to ease parallel and distributed programming. In particular, there are no global variables; functions may only update objects passed to them as **var** (in-out) parameters. Furthermore, as part of passing an object as a **var** parameter, it is effectively *handed off* to the receiving function, and compile-time checks ensure that no further references are made to the object, until the function completes. In particular, the checks ensure that no part of the **var** parameter is passed to any other function, nor to this same function as a separate parameter. This eliminates at compile-time the possibility of aliasing between a **var** parameter and any other object visible to the function. These two additional rules, coupled with the lack of pointers, mean that all parameter evaluation may happen in parallel (e.g. in "`F(G(X), H(Y))`", `G(X)` and `H(Y)` may be evaluated in parallel), and function calls may easily cross address-space boundaries, since the objects are self-contained (with no incoming or outgoing references), and only one function at a time can update a given object.

In the tree-based map example (given in the Appendix), the recursive routine `Count_Subtree` contains a pair of recursive calls which can be safely evaluated in parallel with each other, thanks in part to the ParaSail model which eliminates pointers and global variables.

6. Concurrent Objects

All of the above rules apply to objects that are *not* designed for concurrent access. ParaSail also supports the construction of concurrent objects, which allow lock-free, locked, and queued simultaneous access. These objects are *not* "handed off" as part of parameter passing; concurrent objects provide operations that synchronize any attempts at concurrent access. Three kinds of synchronization are supported. *Lock-free* synchronization relies on low-level hardware-supported operations such as atomic load and store, and compare-and-swap. *Locked* synchronization relies on automatic locking as part of calling a **locked** operation of a concurrent object, and automatic unlocking as part of returning from the operation. Finally, *queued* synchronization is provided, which evaluates a *dequeue condition* upon call (under a lock), and only if the condition is satisfied is the call allowed to proceed, still under the lock. A typical *dequeue condition* might be that a buffer is not full, or that a mailbox has at least one element in it. If the dequeue condition is not satisfied, then the caller is added to a queue. At the end of any operation on the concurrent object that might change the result of the dequeue condition for a queued caller, the dequeue condition is evaluated and if true, the operation requested by the queued caller is performed before the lock is released. If there are multiple queued callers, then they are serviced in turn until there are none with satisfied dequeue conditions.

See the third example in the Appendix, the `Locked_Box`, for an example of a concurrent module.

7. Related Work

There are very few pointer-free languages currently under active development. Fortran 77 [2] was the last of the Fortran series that restricted itself to a pointer-free model of programming. Algol 60 lacked pointers [3], but Algol 68 introduced them [4]. Early versions of Basic had no pointers [5], but modern versions of Basic use pointer assignment semantics for most complex objects [6]. The first versions of Pascal, Ada, Modula, C, and C++ all used pointers for objects that were explicitly allocated on the heap, while still supporting stack-based records and arrays; these languages also required manual heap storage reclamation. The first versions of Eiffel, Java, and C# provided little or no support for stack-based records and arrays, moving essentially all complex objects into the heap, with pointer semantics on assignment, and automatic garbage collection used for heap storage reclamation.

In many cases, languages that originally did not require heavy use of pointers, as they evolved to support object-oriented programming, the use of pointers increased, often accompanied by a reliance on garbage collection for heap storage reclamation. For example, Modula-3 introduced *object* types, and all instances of such types were allocated explicitly on the heap, with pointer semantics on assignment, and automatic garbage collection for storage reclamation [7].

The Hermes language (and its predecessor NIL) was a language specifically designed for distributed processing [8]. The Hermes type system had high-level type constructors, which allowed them to eliminate pointers. As the designer of Hermes explained it, "pointers are useful constructs for implementing many different data structures, but they also introduce aliasing and increase the complexity of program analysis" [9]. Hermes pioneered the notion of *type state*, as well as *handoff* semantics for communication, both of which are relevant to ParaSail, where compile-time assertion checking depends on flow analysis, and handoff semantics are used for passing **var** parameters in a call on an operation.

The SPARK language, a high-integrity subset of Ada with added proof annotations, omits pointers from the subset [10]. No particular attempt was made to soften the effect of losing pointers, so designing semi-dynamic data structures such as trees and linked-lists in SPARK requires heavy use of arrays [11].

Whiley, a new language that has similar goals to ParaSail, has also chosen to avoid pointers and adopt a mutable value semantics [12]. Whiley does not support pervasive parallelism, but rather adopts an explicit *actor* model for concurrency, and allows parameter aliasing in certain contexts. Whiley provides high-level data structuring primitives such as maps, sets, and tuples, rather than adopting the general notion of expandable objects through the use of optional values.

Another relatively new language that is pointer-free is Composita, described in the 2007 Ph. D. thesis of Dr. Luc Bläser from ETH in Zurich [13]. Composita is a component-based language, which uses message passing between active components. Sequences of statements are identified as either *exclusive* or *shared* to provide synchronization between concurrent activities. Composita has the notion of *empty* and *installed* components, analogous to the notion of optional values in ParaSail.

Annotations that indicate an *ownership* relationship between a pointer and an object can provide some of the same benefits as eliminating pointers [14]. AliasJava [15] provides annotations for specifying ownership relationships, including the notion of a *unique* pointer to an object. Guava [16] is another Java-based language that adds *value* types which have no aliases, while still retaining normal *object* types for other purposes. Assignment of value types in Guava involves copying, but they also provide a *move* operation essentially equivalent to that in ParaSail. These approaches, by limiting the possibilities for aliasing, can significantly help in proving desirable properties about programs that use pointers. However, the additional programmer burden of choosing between multiple kinds of pointers or objects based on their aliasing behavior can increase the complexity of such approaches.

One reason given in these papers on aliasing control for not going entirely to a pointer-free, or unique-pointer approach for object-oriented programming, is that certain important object-oriented programming paradigms, such as the Observer pattern [17], depend on the use of pointers and aliasing. ParaSail attempts to provide an existence proof to the contrary of that premise, as do other recent pointer-free languages such as Whiley and Composita. In general, a more loosely-coupled pointer-free approach using container data structures with indices of various sorts, allows the same problem to be solved, with fewer storage management and synchronization issues. For example, the Observer pattern, which is typically based on lists of pointers to observing objects, might be implemented using a pointer-free Publish-Subscribe pattern, which can provide better scalability and easier use of concurrency [18]. In general, pointers are not directly usable in distributed systems, so many of the algorithms adopted to solve problems in a distributed manner are naturally pointer-free, and hence are directly implementable in ParaSail.

Pure functional languages, such as Haskell [19], avoid many of the issues of pointers by adopting immutable objects, meaning that sharing of data creates no aliasing or race condition problems. However, *mostly* functional languages, such as those derived from the ML language [20], include references to mutable objects, thereby re-introducing most of the potential issues with aliasing and race conditions. Even Haskell has found it necessary to introduce special *monads* such as the IO monad to support applications where side-effects are essential to the operation of the program. In such cases, these side-effects need to be managed in the context of parallel programming [21].

Hoare in his 1975 paper on Recursive Data Structures [22] identified many of the problems with general pointers, and proposed a notation for defining and manipulating recursive data structures without the use of pointers at the language level, even though pointers were expected to be used at the implementation level. Language-level syntax and semantics reminiscent of this early proposal have appeared in functional languages, but have not been widely followed in languages with mutable values. Mostly-functional languages such as ML have also more followed the Algol 68 model of explicit references when defining mutable recursive data structures, despite Hoare's many good arguments favoring a pointer-free semantics at the language level. Hoare's notation did not introduce the notion of optional values, but instead relied on types defined by a tagged union of generators, at least one of which was required to not be recursive. ParaSail adopts the optional value approach and allows the set of generators that can be used to create objects to be open-ended, by relying on object-oriented polymorphism over interfaces.

Minimizing use of a global heap through the use of region-based storage management was proposed by Tofte and Talpin [23], implemented in the ML Kit with Regions [24], and refined further in the language Cyclone [25]. Cyclone was not a pointer-free language. Instead, every pointer was associated with a particular region at compile time, allowing compile-time detection of dangling references. A global, garbage-collected heap was available, but local dynamic regions provided a safe, more efficient alternative.

Many functional (or mostly functional) languages have a notion similar to ParaSail's *optional* objects. For example, in Haskell they are called *maybe* objects [19]. In ParaSail, because of its fundamental role in supporting recursive data structures, *optional* is a built-in property usable with every object, component, or type declaration, rather than being an additional level of type. In addition, this approach allows null-ness to be represented without a distinct null object, by ensuring that every type has at least one bit pattern than can be recognizable as the null for that type.

8. Implementation Status and Evaluation

A prototype version of the ParaSail compiler front end and accompanying documentation is available for download [1]. The front end supports nearly all of the language (defined in an accompanying reference manual), and generates instructions for a *ParaSail Virtual Machine* (PSVM). A full multi-threaded interpreter for the PSVM instruction set is built into the front end, and includes a simple interactive Read-Eval-Print Loop for testing. A backend that translates from the PSVM instruction set to a compilable language is under development, with C, Ada, and LLVM assembly language as the initial targets.

The ParaSail front end automatically splits computations up into very light-weight *picothreads,* each representing a potentially parallel sub-computation. The PSVM includes special instructions for spawning and awaiting such picothreads. The PSVM interpreter uses the *work stealing* model [26] to execute the picothreads; work stealing incorporates heavier weight server processes which each service their own queue of picothreads (in a LIFO manner), stealing from another server's queue (in a FIFO manner) only when their own queue becomes empty.

ParaSail adopted a pointer-free model initially to enable easy and safe pervasively parallel programming. However, the early experience in programming in ParaSail with its pointer-free, mutable value semantics, has provided support for the view that pointers are an unnecessary burden on object-oriented programming. The availability of optional values allows the direct representation of tree structures, singly-linked lists, hash tables, and so on in much the same way they are represented with pointers, but without the added complexities of analysis, storage management, and parallelization associated with pointers.

As discussed above, data structures that inherently require multiple paths to the same object, such as doubly-linked lists or general graphs, can be implemented without pointers by using indexing into generalized container structures. Even without the restriction against pointers, it is not uncommon to represent directed graphs using indices rather than pointers, in part because the presence or absence of edges between nodes does not necessarily affect whether the node itself should exist.

A significant advantage of using a container such as a vector to represent a graph, is that partitioning of the graph for the purpose of a parallel divide-and-conquer computation over the graph can be simplified, by using a simple numeric range test on the index to determine whether a given node is within the subgraph associated with a particular sub-computation. As an example, see the `Boundary_Set` operation within the directed graph example in the Appendix. More generally, operations on indices tend to be easier to analyze than those on pointers, including, for example, a proof that two variables contain different indices, as would be needed for a proof of non-aliasing when the indices are used to index into a container.

In our early experiments, programmers familiar with Java or C# have not had trouble understanding and learning to program in ParaSail, thanks in part to its familiar class-and-interface object-oriented programming model. The notion of optional values matches quite directly how pointers work. The fact that assignment is by copy, and there is a separate *move* operation, is a bit of a surprise, but once explained it seems to make sense. The ease of parallel programming and the lack of problems involving undesired aliasing are seen by these early users as valuable benefits of the shift. Perhaps the bigger challenge for some is the lack of global variables in ParaSail. Eliminating global variables seems to require more restructuring than does doing without pointers. It would be possible to allow global *concurrent* objects in ParaSail without interfering with easy parallelization, but these would add complexity to the language and its analysis in other ways.

The primary purpose of our eliminating pointers remains the support of easy, pervasive parallelism. From that point of view, ParaSail is a good showcase. ParaSail programs produce a great deal of parallelism without the programmer having to make any significant effort. Almost any algorithm that is structured as a recursive walk of a tree (see the `Count_Subtree` operation in the tree-based map example in the Appendix), or as a divide and conquer algorithm such as Quicksort (see the `Boundary_Set` operation in the directed-graph example in the Appendix for a similar divide-and-conquer approach), will by default have its recursive calls treated as potentially parallel sub-computations. Monitoring built into the ParaSail interpreter indicates the level of parallelism achieved, and it can be substantial for algorithms not normally thought of as being *embarassingly* parallel. We believe the implicit, safe, pervasive parallelism provided by ParaSail is one of its unique contributions, and this relies on the simplifications made possible by the elimination of pointers and other sources of hidden aliasing.

9. Pointer-Free Object-Oriented Parallel Programming

The pointer-free nature of ParaSail is not just an interesting quirk. Rather, we believe it represents a significantly simpler way to build large object-oriented systems. By itself it simplifies storage management, assignment semantics, and analyzability, and when combined with the elimination of global variables and parameter aliasing, it allows for the easy parallelization of all expression evaluation, and the easy distribution of computations across address spaces, while still supporting directly mutable data structures. As implied by Hoare in [22], pointers are effectively the "goto" of data structuring, and as such, eliminating their use at the language level can bring analogous benefits to data structures and object-oriented programming, as eliminating the "goto" brought to control structures and procedural programming.

Appendix

Pointer-free tree-based map

Here is an example of a pointer-free tree-based map module, showing both the **interface** for the module, and the **class** that defines its implementation.

```
// The following is the interface to a simple map module, which maps
// a key of type Key_Type to a value of type Element_Type.
// External clients of this module see only the interface declarations.
interface TMap<Key_Type is Ordered<>; Element_Type is Assignable<>> is
    op "[]"() -> TMap;  // create an empty TMap

    func Insert(var TMap; Key : Key_Type; Value : Element_Type);
    func Find(TMap; Key : Key_Type) -> optional Element_Type;
    func Delete(var TMap; Key : Key_Type);
    func Count(TMap) -> Univ_Integer;
end interface TMap;

// Here is a possible implementation of this TMap module,
// based on a binary tree.
// The structure of each node of the binary tree
// is defined by the local (pointer-free) Binary_Node interface, which
// declares four components, a Left and Right optional Binary_Node,
// a Key component, and a Value component, which can be null if the Key
// has been (logically) deleted from the Map.
// The Count component of the Tree tracks the number of non-deleted keys
// in the map.  The Count_Subtree operation is used to verify the correctness
// of the Count component, and is provided here to illustrate
// a parallel recursive operation.
// Declarations in a class preceding the exports keyword are private to the
// implementation.

class TMap is

    interface Binary_Node<> is
      // A simple "concrete" binary node structure used to implement a TMap
        var Left : optional Binary_Node;
        var Right : optional Binary_Node;
        const Key : Key_Type;  // Key controls structure of tree
        var Value : optional Element_Type;  // null means was deleted
    end interface Binary_Node;
```

```
  // Root and Count are mutable components of an object defined by the TMap module
  var Root : optional Binary_Node;  // Root of the tree
  var Count := 0;

  // A private operation used to check the correctness of Count maintenance
  func Count_Subtree(Subtree : optional Binary_Node)
    -> Univ_Integer is
      if Subtree is null then
          return 0;
      else
          const SubCount := // these recursive calls are done in parallel
            Count_Subtree(Subtree.Left) + Count_Subtree(Subtree.Right);
          return (Subtree.Value is null? Subcount: Subcount + 1);
      end if;
  end func Count_Subtree;

exports
  op "[]"() -> TMap is  // create an empty TMap
      return (Root => null, Count => 0);
  end op "[]";

  func Insert(var TMap; Key : Key_Type; Value : Element_Type) is
      // Insert Key => Value pair into TMap
      for M => TMap.Root loop
          if M is null then
              // Not already in the map; add it
              M := (Key => Key, Value => Value,
                    Left => null, Right => null);
              TMap.Count += 1;
          else
             case Key =? M.Key of
               [#less] =>
                 continue loop with M.Left;
               [#greater] =>
                 continue loop with M.Right;
               [#equal] =>
                 // Key already in the map
                 if TMap.Value is null then
                     TMap.Count += 1;  // but had been deleted
                 end if;
                 // Overwrite the Value field
                 M.Value := Value;
                 return;
             end case;
          end if;
      end loop;
  end func Insert;

  func Find(TMap; Key : Key_Type) -> optional Element_Type is
      // Find value associated with Key in the TMap; return null if not found
      for M => TMap.Root while M not null loop
          case Key =? M.Key of
            [#less] =>
              continue loop with M.Left;
            [#greater] =>
              continue loop with M.Right;
            [#equal] =>
              // Found it; return the value (which might be null)
              return M.Value;
          end case;
      end loop;
      // Not found in TMap; return null
      return null;
  end func Find;
```

```
    func Delete(var TMap; Key : Key_Type) is
        // Delete Key from the TMap
        for M => TMap.Root while M not null loop
            case Key =? M.Key of
              [#less] =>
                continue loop with M.Left;
              [#greater] =>
                continue loop with M.Right;
              [#equal] =>
                // Found it; if at most one subtree is non-null,
                 // overwrite it; otherwise, set its value field
                // to null (to avoid a more complex re-balancing).
                if M.Value not null then
                    TMap.Count -= 1; // Decrement unless already deleted.
                end if;
                if M.Left is null then
                    // Move right subtree into M
                    M <== M.Right;
                elsif M.Right is null then
                    // Move left subtree into M
                    M <== M.Left;
                else
                    // Cannot immediately reclaim node;
                    // set value field to null instead.
                    M.Value := null;
                end if;
            end case;
        end loop;
        // Not found in the map
    end func Delete;

    func Count(TMap) -> Univ_Integer is
        // Return count of non-deleted keys
        // Verify that Count has been maintained properly
        // (ParaSail assertions use {})
        {Count_Subtree(TMap.Root) == TMap.Count}
        return TMap.Count;
    end func Count;
end class TMap;
```

Pointer-Free Directed Graph

Here is an example of a pointer-free directed graph, represented as a vector of nodes, each with a Predecessor and Successor set of node-ids to represent edges of the graph. Preconditions, postconditions, and assertions are enclosed in braces ({}) in ParaSail, reminiscent of Hoare logic.

```
interface DGraph<Element is Assignable<>> is
  // Interface to a (pointer-free) Directed-Graph module
    type Node_Id is new Integer<1..10**6>;
      // A unique id for each node in the graph
    type Node_Set is Countable_Set<Node_Id>;
      // A set of nodes

    func Create() -> DGraph;
      // Create an empty graph
    func Add_Node(var DGraph; Element) -> Node_Id;
      // Add a node to a graph, and return its node id
    func Add_Edge(var DGraph; From, To : Node_Id)
      {From in DGraph.All_Nodes(); To in DGraph.All_Nodes()};
      // Add an edge in the graph
    op "indexing"(ref DGraph; Node_Id)
      {Node_Id in DGraph.All_Nodes()}
      -> ref Element;
      // Return a reference to an element of the graph
    func Successors(ref const DGraph; Node_Id) -> ref const Node_Set
      {Node_Id in DGraph.All_Nodes()};
      // The set of successors of a given node
    func Predecessors(ref const DGraph; Node_Id) -> ref const Node_Set
      {Node_Id in DGraph.All_Nodes()};
      // The set of predecessors of a given node

    func All_Nodes(DGraph) -> Node_Set;
      // The set of all nodes
    func Roots(DGraph) -> Node_Set;
      // The set of all nodes with no predecessor
    func Leaves(DGraph) -> Node_Set;
      // The set of all nodes with no successor
end interface DGraph;

class DGraph is // Class defining the Directed-Graph module
    interface Node<> is
      // Local definition of Node structure
        var Elem : Element;
        var Succs : Node_Set;
        var Preds : Node_Set;
    end interface Node;

    var G : Vector<Node>;
      // The vector of nodes, indexed by Node_Id

    func Boundary_Set(DGraph; Nodes : Countable_Range<Node_Id>;
      Want_Roots : Boolean) -> Node_Set is
      // Recursive helper for exported Roots and Leaves functions
        const Len := Length(Nodes);
        case Len of
          [0] =>
            return [];
          [1] =>
            if Want_Roots?
                Is_Empty(Predecessors(DGraph, Nodes.First)):
                Is_Empty(Successors(DGraph, Nodes.First))
            then
                // This is on the desired boundary
                return [Nodes.First];
            else
                // This is not on the desired boundary
                return [];
            end if;
          [..] =>
            // Parallel recursive divide and conquer
            const Half_Way := Nodes.First + Len / 2;
            return
              Boundary_Set(DGraph,
                Nodes.First ..< Half_Way, Want_Roots) |
              Boundary_Set(DGraph,
                Half_Way .. Nodes.Last, Want_Roots);
        end case;
    end func Boundary_Set;

  exports

    func Create() -> DGraph is
      // Create an empty graph
        return (G => []);
    end func Create;

    func Add_Node(var DGraph; Element) -> Node_Id is
      // Add a node to a graph, and return its node id
        DGraph.G |= (Elem => Element, Succs => [], Preds => []);
        return Length(DGraph.G);
    end func Add_Node;

    op "indexing"(ref DGraph; Node_Id) -> ref Element is
      // Return a reference to an element of the graph
        return DGraph.G[Node_Id].Elem;
    end op "indexing";

    func Add_Edge(var DGraph; From, To : Node_Id) is
      // Add an edge in the graph
        DGraph.G[From].Succs |= To;
        DGraph.G[To].Preds |= From;
    end func Add_Edge;

    func Successors(ref const DGraph; Node_Id) -> ref const Node_Set is
      // The set of successors of a given node
        return DGraph.G[Node_Id].Succs;
    end func Successors;

    func Predecessors(ref const DGraph; Node_Id) -> ref const Node_Set is
      // The set of predecessors of a given node
        return DGraph.G[Node_Id].Preds;
    end func Predecessors;

    func All_Nodes(DGraph) -> Node_Set is
      // The set of all nodes
        return 1 .. Length(DGraph.G);
    end func All_Nodes;

    func Roots(DGraph) -> Node_Set is
      // The set of all nodes with no predecessor
        return Boundary_Set
          (DGraph, 1 .. Length(DGraph.G), Want_Roots => #true);
    end func Roots;

    func Leaves(DGraph) -> Node_Set is
      // The set of all nodes with no successor
        return Boundary_Set
          (DGraph, 1 .. Length(DGraph.G), Want_Roots => #false);
    end func Leaves;
end class DGraph;
```

Concurrent locked-box module

Here is an example of a concurrent module, using **locked** and **queued** operations. The Locked_Box contains a single component into which a value of type Content_Type may be stored. The Set_Content and Content operations provide simple, **locked** access to the content of the box, and allow for **null** values. The Put and Get operations deal only with non-null values, with the caller of Put being **queued** until the box is empty before allowing a new value to be stored, and the caller of Get being **queued** until the box has a non-null value, and then returning that value and setting the box back to empty (null).

```
concurrent interface Locked_Box<Content_Type is Assignable<>> is
    func Create(C : optional Content_Type) -> Locked_Box;
        // Create a box with the given content

    func Set_Content(locked var B : Locked_Box; C : optional Content_Type);
        // Set content of box
    func Content(locked B : Locked_Box) -> optional Content_Type;
        // Get a copy of current content

    func Put(queued var B : Locked_Box; C : Content_Type);
        // Wait for the box to be empty (i.e. null)
        // and then Put something into it.
    func Get(queued var B : Locked_Box) -> Content_Type;
        // Wait until content is non-null,
        // then return it, leaving it null.
end interface Locked_Box;
```

```
concurrent class Locked_Box is
    var Content : optional Content_Type;  // Content might be null
  exports
    func Create(C : optional Content_Type) -> Locked_Box is
        // Create a box with the given content
        return (Content => C);
    end func Create;

    func Set_Content(locked var B : Locked_Box; C : optional Content_Type) is
        // Set content of box
        B.Content := C;
    end func Set_Content;

    func Content(locked B : Locked_Box)
      -> optional Content_Type is
        // Get a copy of current content
        return B.Content;
    end func Content;

    func Put(queued var B : Locked_Box; C : Content_Type) is
      queued until B.Content is null then
        // Wait for the box to be empty (i.e. null)
        // and then Put something into it.
        B.Content := C;
    end func Put;

    func Get(queued var B : Locked_Box)
      -> Result : Content_Type is
      queued while B.Content is null then
        // Wait until content is non-null,
        // then return it, leaving it null.
        Result <== B.Content;
    end func Get;

end class Locked_Box;
```

.

References

[1] S. Tucker Taft, *Designing ParaSail: A New Programming Language,* 2009, http://parasail-programming-language.blogspot.com (retrieved 8/10/2012).

[2] Ansi x3.9-1978. *American National Standard – Programming Language FORTRAN*. American National Standards Institute, 1978, http://www.fortran.com/fortran/F77_std/rjcnf.html (retrieved 8/10/2012).

[3] Peter Naur *et al*, *Revised Report on the Algorithmic Language Algol 60,* 1963 http://www.masswerk.at/algol60/report.htm (retrieved 8/10/2012)

[4] C.H. Lindsey, "A History of Algol 68," *Proceedings of HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pp 97-132, ACM New York, NY, 1993.

[5] J.G. Kemeny and T.E. Kurtz, *BASIC, 4th Edition,*Trutees of Dartmouth College, 1968, http://www.bitsavers.org/pdf/dartmouth/BASIC_4th_Edition_Jan68.pdf (retrieved 8/10/2012).

[6] Microsoft, *Visual Basic Concepts -- Programming with Objects*, http://msdn.microsoft.com/en-us/library/aa716290.aspx (retrieved 8/10/2012).

[7] L. Cardelli *et al*, *Modula-3 Report (revised)*, http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-52.pdf (retrieved 8/10/2012).

[8] R. Strom, "Hermes: An Integrated Language and System for Distributed Programming," *Proceedings of the IEEE Workshop on Experimental Distributed Systems,* pp. 75--82, 1990 http://researcher.ibm.com/files/us-bacon/Strom90Hermes.pdf (retrieved 9/27/2012).

[9] *ibid*., p. 80.

[10] R. Chapman and P. Amey, *SPARK-95 – The SPADE Ada Kernel (including RavenSPARK),* Altran-Praxis, 2008 http://www.altran-praxis.com/downloads/SPARK/technicalReferences/SPARK95_RavenSPARK.pdf (retrieved 8/10/2012).

[11] P. Thornley, *SPARKSure Data Structures*, 2009, http://www.sparksure.com/resources/SPARK_Data_Structures_10_09.zip (retrieved 8/10/2012).

[12] D. Pearce, "Whiley overview," http://whiley.org/about/overview/ (retrieved 9/28/2012).

[13] L. Bläser, "A Component Language for Pointer-Free Programming," Ph. D. Thesis, ETH Zurich, 2007 http://e-collection.library.ethz.ch/eserv/eth:30090/eth-30090-02.pdf (retrieved 9/28/2012)

[14] D. Clarke. "Object Ownership & Containment," Ph.D. Thesis, University of New South Wales, Australia, July 2001 http://people.cs.kuleuven.be/~dave.clarke/papers/thesis.ps.gz (retrieved 9/28/2012).

[15] J. Aldrich *et al,* "Alias Annotations for Program Understanding," *OOPSLA'02,* pp. 311-330, Nov. 4-8, 2002, Seattle, WA

http://archjava.fluid.cs.cmu.edu/papers/oopsla02.pdf (retrieved 9/28/2012).

[16] D. Bacon *et al,* "Guava: A Dialect of Java without Data Races," *OOPSLA '00,* pp. 382-400, Minneapolis, MN, 2000 http://researcher.watson.ibm.com/researcher/files/us-bacon/Bacon00Guava.pdf (retrieved 9/30/2012).

[17] E. Gamma, *et al, Design Patterns*, Addison-Wesley, 1994.

[18] G. Hohpe *et al, Enterprise Integration Patterns* website, "JMS Publish/Subscribe Example," 2003 http://www.eaipatterns.com/ObserverJmsExample.html (retrieved 9/28/2012).

[19] S. Marlow, *Haskell 2010 Language Report*, 2010, http://www.haskell.org/onlinereport/haskell2010/ (retrieved 8/10/2012).

[20] R. Harper, *Programming in Standard ML*, Carnegie Mellon University, 2005, http://www.cs.cmu.edu/~rwh/smlbook/book.pdf (retrieved 8/10/2012).

[21] S. Marlow *et al,* "A Monad for Deterministic Parallelism," *Haskell '11 Proceedings of the 4th ACM symposium on Haskell*, pp. 71-82, ACM New York, NY, 2011.

[22] C.A.R. Hoare, C.B. Jones (Ed.), "Recursive Data Structures," *Essays in Computing Science,* pp. 217-244, Prentice-Hall International (UK), 1989 http://dl.acm.org/citation.cfm?id=63445&bnc=1 (retrieved 9/30/2012).

[23] M. Tofte and J.-P. Talpin, "Implementing the call-by-value lambda calculus using a stack of regions," *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* pp. 188-201, ACM Press, 1994.

[24] M. Tofte and J.-P. Talpin, "Region-based memory management," *Information and Computation*, 132.2, pp. 109-176, 1997 http://www.irisa.fr/prive/talpin/papers/ic97.pdf (retrieved 9/27/2012).

[25] D. Grossman *et al*, "Region-Based Memory Management in Cyclone," *PLDI '02 Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pp. 282 – 293, ACM New York, NY, 2002 http://www.eecs.harvard.edu/~greg/cyclone/papers/cyclone-regions.pdf

[26] R. Blumofe and C. Leisersen, "Scheduling Multithreaded Computations by Work Stealing," *Journal of the ACM*, 46.5, pp. 720-748, Sep. 1999 http://supertech.csail.mit.edu/papers/ft_gateway.pdf (retrieved 9/29/2012).