Designing ParaSail – Parallel Specification and Implementation Language

S. Tucker Taft

SofCheck, Inc, 11 Cypress Drive, Burlington, MA 01803 USA.; Tel: +1 781 850 7068; email: stt@sofcheck.com

Abstract

This (edited) blog follows the trials and tribulations of designing a new programming language designed to allow productive development of parallel, high-integrity (safety-critical, high-security) software systems. The language is named "ParaSail" for Parallel, Specification and Implementation Language.

Keywords: parallel programming language, high-integrity software, formal methods.

Introduction

This article is an extract from a blog started in September 2009 describing the process of designing ParaSail, a new programming language for high-integrity, parallel programming. If you wish to read the full blog, please visit [1]. We have left out large parts of the blog, which in its entirety would run to about 35 pages. We have included editorial comments in *"[...]"* both to describe what we are leaving out, and to describe changes since a given blog entry was written. We have also fixed the examples to use the latest ParaSail syntax, to avoid creating confusion. Note that the online blog entries have not been edited, so if you read those, beware that some of the syntactic details have been changed over time.

September 2009

Why design a new programming language?

So why would anyone want to design a new programming language? For some of us who have the bug, it is the ultimate design project. Imagine actually creating the language in which you can express yourself. But there is another reason. I have been in the software business for over 40 years, and despite everything that might have been said to the contrary, I still believe that a well-designed programming language can result in more productive programmers building higher quality software. In the particular area of high-integrity software, including both safety-critical software and high-security software, there is all the more reason to use the very best programming language you can, because the problems you are trying to solve and the level of quality required is at the very limits of what can be accomplished.

This new language is meant to address the goals of producing inherently safe and secure software, while taking advantage of the wider availability of true parallel processing in the form of multi-core chips. It is intended to promote a formal approach to software, where the program text includes pre- and postconditions, liberal use of assertions and invariants, etc., with tool-supported proof of correctness with respect to the formal annotations.

The language is tentatively named **ParaSail**, for Parallel Specification and Implementation Language. I would have spelled it "ParaSAIL" but for the danger of confusion with the original Stanford AI Language, "SAIL" [2], and its more modern follow-on "MAINSAIL" (for Machine Independent SAIL). I don't mind making the connection with SAIL, as it was a very interesting language in its day, and MAINSAIL remains worth a look today. ParaSail is a completely new language, but it steals liberally from other programming languages, including the ML series (SML, CAML, OCAML, etc.), the Algol/Pascal family (Algol, Pascal, Ada, Modula, Eiffel, Oberon, etc.), the C family (C, C++, Java, C#), and the region-based languages (especially Cyclone). Perhaps one significant deviation from the excellent baseline established by SAIL, ML, Eiffel, Java, etc. is that ParaSail is intended to avoid "fine-granule" garbage collection in favor of stack and region-based storage management.

Why blog about designing a programming language?

So why did I decide to start this blog? Designing a new language is a long process, but it is hard to find someone who is willing to sit down and discuss it. Those who like to design languages generally have their own strong biases, and the discussions tend to be more distracting than satisfying, because there are so many good answers to any interesting language design question. Those who don't have any interest in designing a new language tend to lack the patience to even talk about it, as they believe we already have more than enough programming languages, and all we need are better tools, better processes, and better training to be more productive and to achieve higher quality.

So writing a blog seems like a nice way to record the process, to perhaps get some feedback (though that may be optimistic), and to hopefully make progress by being forced to actually get the ideas down onto "paper."

That's probably enough meta-discussion for now. In the next post I plan to dive into the technical issues.

ParaSail language themes and philosophy

So what will make ParaSail an interesting programming language? What is the philosophy behind the language? ParaSail tries to minimize implicit operations, implicit parameters, implicit dynamic binding (virtual function calls), implicit initializations, implicit conversions, etc. This is both in the name of clarity for the human reader, and in the name of formal testability and verifiability. ParaSail uses a small number of concepts to represent all of the various composition mechanisms such as records, packages, classes, modules, templates, capsules, structures, etc. Arrays and more general containers are treated uniformly.

On the other hand, ParaSail allows many things to proceed in parallel by default, effectively inserting implicit parallelism everywhere. Parameter evaluation is logically performed in parallel. The language disallows uses that would make the result depend on the order or concurrency of parameter evaluation. The iterations of a **for** loop are by default executed in parallel. Explicit ordering must be specified if it is required by the algorithm. Even sequential statements are essentially converted into a data-flow based DAG which is then evaluated in parallel in so far as possible. In all cases, the language disallows code that could result in race conditions due to inadequately synchronized access to shared data, either by using per-thread data, structured safe synchronization, or a handoff semantics (similar to that of linear types, distributed languages like Hermes [3], or the UVM virtual memory system). …

Modules in ParaSail

…ParaSail has only two kinds of modules: interface modules and class modules. A class module implements one or more interface modules. Perhaps a class ought to be called an implementation module, but that is an awfully long word, and in many ways a class module matches what languages like Simula and C++ and Java and C# and Eiffel call a class. In general ParaSail tries to use familiar terminology in familiar ways. Of course there is danger of confusion, but hopefully the benefits of familiarity outweigh the dangers of confusion. *[We have subtly altered the vocabulary since this was written. We now refer to the interface* of *a module, and the class that* defines *the module. So there is really only one kind of module, but like a Modula module or an Ada package, it has a part that declares the external interface for the module, and a part that defines the internals of the module. The "class" part is optional. It need not be provided if the module is* ***abstract****, or if there are no operations in the interface, merely components.]*

Both interface and class modules are parameterized, essentially generic templates, as used in Ada, C++, Java, etc. Because of their wide-spread adoption, ParaSail uses "<...>" for generic parameters. While we are talking about notation, ParaSail uses "(...)" for normal function/procedure parameters, uses "[...]" for selecting from an array or similar container. ParaSail uses words for bracketing control flow (e.g. "if ... then ... else ... end if;"). ParaSail uses "{...}" for annotations such as constraints, assertions, pre/post-conditions, etc. This is intended to be familiar from Hoare logic, which uses the notation "{P} S {Q}" to represent a statement S with precondition P and postcondition Q.

Interface modules have the following basic syntax:

```
interface name < generic_parameters >
is
  interface_items
end interface name;
```

The interface_items comprise a sequence of function, procedure, and nested interface specifications. In addition constants and variables may be declared, but these are considered equivalent to corresponding functions. … In addition, operators may be declared. Operators are essentially functions with special syntax for calling them.

As subtly implied above, an interface defines a type, and within the interface module, the interface's name represents that type. A type is essentially an interface that has been bound to a set of actual generic parameters. There are also subtypes in ParaSail, which are types with some additional constraints (such as a range limitation), specified inside "{...}".

Generic parameters are themselves either types derived from specified interfaces, or static constants used to parameterize the implementation of the interface, and/or select among different implementations of the same interface. A generic array interface, with its generic parameter list, might look like this:

```
interface Array < Element_Type is Assignable<>;
           Index_Type is Discrete<>>
is
  function First(Arr : Array) -> Index_Type;
  function Last(Arr : Array) -> Index_Type;
  function Element(Arr : ref Array;
    Index : Index_Type {Index in First(Arr)..Last(Arr)})
      -> ref Element_Type;
  function Create_Array(First, Last : Index_Type)
      -> Result:Array
        {First(Result) = First; Last(Result) = Last};
  ...
end interface Array;
```

Note that there is no implicit parameter in ParaSail (identified as `this` or `self` in many object-oriented languages). All parameters are explicit….

More on ParaSail interfaces

Continuing with the above example… To make indexing into an Array look more familiar, we might want to use the "[]" operator instead of the function Element:

```
operator "[]"(Arr : ref Array;
  Index : Index_Type {Index in First(Arr)..Last(Arr)})
    -> ref Element_Type;
```

Now we could use this interface roughly as follows:

```
var Names: Array<String, Student_ID> :=
  Create_Array(First => First_Student_ID,
          Last => Last_Student_ID);
Names[ID_Of_Mike] := "Mike";
```

This kind of thing should look pretty familiar.

A few things to note: There is no separate notion of a constructor. A function that returns an object of the type defined by the enclosing interface is effectively a constructor. A function that returns a **ref** is not a constructor, but may be used on the left-hand side of an assignment, provided the **ref** parameter(s) to the function can be used on the left-hand side. Parameters are by default read-only, but **ref var** parameters can be updated. The result of a function is specified after a "->". If the function has multiple results, they are enclosed in parentheses in the same way as the (input) parameters:

```
function Two_Outputs(X : Input_Type) ->
      (Y : Output_Type1; Z : Output_Type2);
```

Once we start talking about statements, we will see that in ParaSail, as in C++ and Java, declarations and executable statements may be interspersed. All declarations start with a reserved word, such as **var**, **const**, **interface**, **type**, etc. so they are easy to distinguish from assignment statements, procedure calls, etc.

Next time we will discuss how a class implements an interface.

How a ParaSail class implements its interface

In ParaSail, a class implements an interface. Unless an interface is marked as **abstract**, there is required to be a class with the same name as the interface, which provides the default implementation of the interface. There may be multiple implementations of an interface. Some may be specializations of the default. Specializations specify certain restrictions on the generic parameters, along with a preference level, which allows the compiler to choose the appropriate implementation automatically if not specified explicitly.

Here is a class implementing the interface Array shown in an earlier post:

```
class Array < Element_Type is Assignable<>;
          Index_Type is Discrete<>>
is
  const First : Index_Type;
  const Last : Index_Type;
  function Length(Arr : Array) -> Integer is
    return Last-First+1;
  end function Length;
  var Data : Basic_Array<Element_Type>;
exports
  function First(Arr : Array) -> Index_Type is
    return Arr.First;
  end function First;
  …
  operator "[]"(Arr : ref Array;
    Index : Index_Type {Index in First(Arr)..Last(Arr)})
      -> ref Element_Type is
    return Data[Index-First];
  end operator "[]";
  function Create_Array(First, Last : Index_Type)
      -> Result:Array
         {First(Result) = First and Last(Result) = Last}
  is
    return (First => First, Last => Last,
      Data => Create(Last-First+1));
  end function Create_Array;
  ...
end class Array;
```

A class must include implementations for each operation in its interface, plus any number of local declarations as necessary to implement the exported operations. The exported declarations inside a class follow the word **exports**. Declarations preceding **exports** are local to the class. Even though the interface associated with the class implies what operations are exported, we make the distinction explicit in the class itself, since changing the specification of an exported declaration has a much larger implication than does changing that of a local declaration. Also, by segregating the exported declarations at the end, we make it easier to find them. Finally, because a class might implement interfaces other than its own, the exported declarations allow it to fulfill the set of operations of other interfaces. C uses "extern" vs. "static" to make a similar distinction. Java uses "public" vs. "private". …

Resolving names in ParaSail

In ParaSail, the names exported by an interface can in most cases be used without any explicit qualification of the interface from which they came. Hence, continuing with our Array interface example, we can generally write "First(Arr)" without having to specify which "First" function we mean, presuming we know the type of Arr. If qualification is necessary, the "::" notation of C++ is used, such as "Array::First(Arr)". ParaSail reserves "." for selecting a component of a composite object, such as "Obj.Field", or as a way of indicating an operation is defined within the module defining the type of the object, such as “Arr.First()”, which is equivalent to “<type_of_Arr>::First(Arr)”.

…

ParaSail extension, inheritance, and polymorphism

ParaSail fully supports object-oriented programming, including interface and class extension/inheritance/reuse, and both static (compile-time) and dynamic (run-time) polymorphism.

Interface extension is the most straightforward. One interface can be defined to extend another, meaning that it inherits all of the operations and generic parameters of some existing interface, and optionally adds more of each:

```
interface Extensible_Array extends Array is
   operator "[]"(Arr : ref Extensible_Array;
      Index : Index_Type {Index >= First(Arr)})
       -> ref Element_Type {Last(Arr) >= Index};
end interface Extensible_Array;
```

Here we have essentially the same operation, but now the array automatically grows (at only one end) if the indexing operation is applied to it with an index that is greater than the prior value of Last(Arr). Note that the generic parameters for Extensible_Array are all inherited as is from Array.

The operations that are not redeclared in the new interface are inherited from Array, but with each occurrence of Array replaced systematically with Extensible_Array.

Unless Extensible_Array is declared as an **abstract** interface, there must be a corresponding Extensible_Array **class** that provides its default implementation. The Extensible_Array class might be defined as an extension of the Array interface (and indirectly its associated class), but it need not be. *[We now require that a class **extends** another module if and only if its interface also **extends** that module. The reserved word **implements** is used (rather than **extends**) when an interface inherits from another module's interface but does* not *inherit the code from the other module's class. An interface may specify any number of other modules after the reserved word **implements** but only one after **extends**.]* … Here is what it would look like if it were an extension of the Array interface.

```
class Extensible_Array extends Parent: Array
is
  ...
exports
  operator "[]"(...) -> ... is ... end operator "[]";
  function Create_Array(...) -> Extensible_Array is ...
  end function Create_Array;
end class Extensible_Array;
```

Local operations and objects could be defined as needed, and exported operations could be redefined as appropriate. An object of the interface being extended (the underlying interface) is created as a local variable of the class (this variable is the *underlying object*). A name can be given to the underlying object -- we use "Parent" above.

Note that the class is extending an *interface*, rather than directly another class, and in fact any implementation of the interface can later be optionally specified when using Extensible_Array. Effectively the actual class to use as the implementation of the underlying interface becomes another optional generic parameter. If unspecified, it defaults to the default implementation of the underlying interface.

For those operations of the interface that are not provided explicitly in the class, an inherited operation is provided, based on the corresponding operation of the underlying interface. The implementation of this inherited operation invokes the corresponding operation of the Array interface, passing it the underlying Array object of each operand of type Extended_Array. …

Although an interface may *[implement]* any number of other interfaces, a class can only extend one interface. Of course it may have local variables of many different interfaces, but only one of them is the underlying object, and only inherited operations corresponding to operations on the underlying interface will be automatically provided.

One important thing to note about how ParaSail implementation inheritance works. It is a completely *black box*. Each implicitly provided inherited operation calls the corresponding operation of the underlying interface, passing it the underlying objects of any operands of the type being defined. The underlying interface operation operates only on the underlying object(s), having no knowledge that it was called "on behalf" of some extended class. If the underlying operation calls other operations, they too are operating only on the underlying object(s). There is no "redispatch" on these internal calls, so the extended class can treat these underlying operations as black boxes, and not worry that if it explicitly defines some operations while inheriting others, that that might somehow interact badly with how the underlying operations are implemented.

Which brings us to *polymorphism*. Static, compile-time polymorphism has already been illustrated, through the use of generic parameters, and the name resolution rules. We also see it here in that the particular class implementing the underlying interface for an extended class can also vary depending on the particular instantiation of the extended class. Although we didn't mention the syntax for that, it makes sense for it to use the **extends** keyword as well at the point when we are providing the actual generic parameters:

```
var XArr : Extended_Array
  extends Sparse_Array := Create_Array(...);
```

Now what about dynamic, run-time polymorphism? When do we actually need it? A classic example is that of a heterogeneous tree structure, where all of the nodes in the tree are extensions of some basic Tree_Node type, but each is specialized to carry one particular kind of information or another. Static polymorphism is great for creating general purpose algorithms and data structures, but they are inevitably homogeneous to some degree. The algorithm manipulates numbers all of the same precision, or the data structure holds elements all of the same type. With dynamic polymorphism, heterogeneity rules.

In ParaSail, dynamic polymorphism is indicated by appending a "+" to a type name *[(e.g.* `Tree_Node+`*)]*. What this means is that the corresponding object, parameter, or result might be of a type coming from any extension of the given type's interface, with the proviso that the generic parameters inherited from this original interface have the same bindings for the two types. That is important because, without that, the operations shared between the two types would not necessarily take the same types of parameters. …

October 2009

ParaSail pass-by-reference …

ParaSail has no global variables

ParaSail has no global variables. So how does that work? Global variables have long been recognized as trouble makers in the software world, and they get much worse in a language with lots of parallelism. But can we eliminate global variables completely? ParaSail tries to do that by eliminating any notion of statically allocated variables. In C or C++, global variables are those marked static or extern, or in a language with modules/packages like Ada, global variables are those declared at the module level. Although their declaration might be hidden, such variables still represent global, variable state, and still create trouble.

In ParaSail, interface modules are generic types, and class modules implement interfaces. That is, after providing generic parameters to an interface, you end up with a type. Any variable declared in a class is what is often called an *instance variable*, that is, there is one per object of the type defined by the class. There is nothing corresponding to what are called *class variables*. In other words, the variables declared in a class will always be effectively fields/components of some enclosing object. You can of course also have local variables inside the body of a procedure or function, but these are never statically allocated; they are stack-resident variables only (automatic in C parlance).

…

ParaSail's implicit parallelism

As mentioned in the original overview of ParaSail, implicit parallelism is at the heart of ParaSail (and of its name!). Every procedure/function call with multiple parameters involves implicit parallelism, in that all of the parameters are evaluated in parallel. Handoff semantics is used for writable parameters, meaning that a (non-concurrent) object that is writable by one parameter evaluation, isn't available to any other parameter evaluation. Operations on concurrent objects are not ordered.

Furthermore, in a sequence of statements, the only default limitation on parallelism is that imposed by operations on non-concurrent objects. Parallelism can be inhibited by using ";;" instead of merely ";" to separate two statements. This forces sequencing for operations on concurrent objects, which would otherwise not have any specified order. By contrast, "||" can be used to indicate that parallelism is expected, and it is an error if there are conflicting operations on non-concurrent objects in the statements on either side of the "||". That is, two statements separated by "||" are executed effectively in parallel, just as though there were embedded within two different parameter evaluations to a single procedure or function call. *[We have changed the precedence for these operators since this blog entry, and added another operator "**then**" (equivalent to ";;") with lowest precedence, "||" higher, and ";" and ";;" at the highest precedence (bind most tightly).]* For example:

```
   X := F(A, B);
   Y := G(B, C);
then
   Z := H(R, S);
|| Z2 := Q(R, T);
```

So in the above, the assignments to X and Y are ordered only if that would be required by the shared use of B (at least one of them has B as a writable parameter). The assignments to Z and Z2 occur in parallel, and it would be an error if the shared use of R is unsafe (e.g. because R is writable by one or both, and is not a concurrent object). The assignments to X and Y will complete before beginning the assignments to Z and Z2 (though of course, a "very intelligent" compiler might still find parts that can be executed in parallel because they can't possibly affect one another). In all cases the evaluations of the parameters to a single procedure/function call happen in parallel. …

Implicit parallelism also shows up in ParaSail loops. The iterations of a **for** loop in ParaSail are executed as though each iteration were separated from the next by a ";", but with no particular ordering on the iterations. A particular ordering may be specified by using the **forward** or **reverse** keyword, which effectively puts a ";;" between the iterations. Alternatively, the **concurrent** keyword may be used to indicate that the iterations should be effectively separated by "||", in which case it is an error if there are any conflicts due to operations on a non-concurrent variable. …

Using "=?" in ParaSail to implement "==", "!=", "<=", ">=", etc.

Here is a simple ParaSail feature. It is often the case that a user-defined type will have its own definition for equality and comparison. However, it is always a bit of an issue to make sure that "==" and "!=" are complements, and similarly for "<" and ">=", ">" and "<=". ParaSail skirts this issue by requiring the user to define only one operator, "=?", which returns a value from the enumeration #less, #equal, #greater, #unordered. The comparison and equality operators are defined in terms of "=?" (which is pronounced "compare"), in the natural way…

November 2009

ParaSail region-based storage management ...

ParaSail concurrent interfaces

A ParaSail interface can be marked as **concurrent**, meaning that an object of a type produced by instantiating such a concurrent interface can be accessed concurrently by multiple parallel threads. We use the term *concurrent type* for a type produced by instantiating a concurrent interface, and *concurrent object* for an object of a concurrent type.

Both *lock-based* and *lock-free* synchronization techniques can be used to coordinate access to a concurrent object. In addition, *operation queues* are provided to hold operation requests that cannot be serviced until the concurrent object attains a desired state, as specified by the *dequeue condition*. When a thread initiates an operation request, if it can't be performed immediately because the condition is False, the thread is blocked until the request gets dequeued and the associated operation is completed, or the operation request is canceled for some reason, such as a timeout. …

With a lock-based concurrent interface, one of the operands of a procedure or function of the interface … can be marked indicating it should be **locked** upon call. When calling the operation, the operand's lock will be acquired automatically upon entry to the operation, and the lock will be released upon return. Alternatively, one operand of a concurrent interface operation may be marked **queued**, meaning that the *dequeue condition* must be true before the operation will be performed, and when it is performed it will be locked automatically as well.

Within the concurrent class implementing a lock-based concurrent interface, the components of an operand marked **locked** or **queued** may be referenced directly, knowing that the access is single-threaded at that point. If there are other (non-locked/queued) operands of the associated concurrent type, their non-concurrent components may *not* be referenced by the operation. An operation with a **queued** operand has the further assurance that at the point when the operation starts, the dequeue condition is True. The dequeue condition for a given operation may depend on the values of the non-concurrent parameters, plus the internal state of the (concurrent) operand marked **queued**.

Within the concurrent class implementing a *lock-free* concurrent interface, all data components must themselves be concurrent objects, since no locking is performed on entry to the operations of the class. Typically these components will be of low-level concurrent types, such as a single memory cell that supports atomic load, atomic store, and atomic compare-and-swap (CAS), or a somewhat higher-level concurrent type that supports multi-word-compare-and-swap (MCAS). The ParaSail standard library provides a small number of such lower-level concurrent types to support implementing lock-free concurrent interfaces. The higher level **queued** operations are implemented using a lower-level race-free and lock-free concurrent type similar to a private semaphore, which can be used directly if desired.

Here is an example of a concurrent interface and corresponding concurrent class for implementing a simple bounded buffer:

```
concurrent interface Bounded_Buffer
  <Element_Type is Assignable<>;
   Index_Type is Integer<>> is
    function Create_Buffer(
      Max_In_Buffer : Index_Type {Max_In_Buffer > 0})
      -> Result : Bounded_Buffer;
      // Create buffer of given capacity
    procedure Put(Buffer : queued Bounded_Buffer;
      Element : Element_Type);
      // Add element to bounded buffer;
      // remain queued until there is room
      // in the buffer.
    function Get(Buffer : queued Bounded_Buffer)
      -> Element_Type;
      // Retrieve next element from bounded buffer;
      // remain queued until there is an element.
end interface Bounded_Buffer;

concurrent class Bounded_Buffer is
    const Max_In_Buffer : Index_Type
      {Max_In_Buffer > 0};
    var Data : Array<Element_Type, Index_Type> :=
      Create_Array(1, Max_In_Buffer);
    var Next_Add : Index_Type
      {Next_Add in 1..Max_In_Buffer} := 1;
    var Num_In_Buffer : Index_Type
      {Num_In_Buffer in 0..Max_In_Buffer} := 0;
  exports
    function Create_Buffer(
      Max_In_Buffer : Index_Type {Max_In_Buffer > 0})
      -> Bounded_Buffer is
        return (Max_In_Buffer => Max_In_Buffer);
    end function Create_Buffer;
    procedure Put(Buffer : queued Bounded_Buffer;
      Element : Element_Type)  is
      queued until
        Buffer.Num_In_Buffer < Buffer.Max_In_Buffer then
        Buffer.Data[Buffer.Next_Add] := Element;
        // Advance to next element, cyclically.
        Buffer.Next_Add :=
         (Buffer.Next_Add mod Buffer.Max_In_Buffer) + 1;
        Buffer.Num_In_Buffer += 1;
    end procedure Put;
    function Get(Buffer : queued Bounded_Buffer)
      -> Element_Type is
      queued until Buffer.Num_In_Buffer > 0 then
         return Buffer.Data[
           ((Buffer.Next_Add - Buffer.Num_In_Buffer - 1)
             mod Buffer.Max_In_Buffer) + 1];
    end function Get;
end class Bounded_Buffer;
```

This concurrent interface has one constructor, plus two queued operations, Put and Get. The dequeue conditions are provided as part of the implementation of an operation with a queued operand. Here the dequeue conditions are Buffer.Num_In_Buffer < Buffer.Max_In_Buffer for Put, and Buffer.Num_In_Buffer > 0 for Get. These dequeue conditions ensure that when the operations are performed, they can proceed without an error. …

December 2009

ParaSail end-of-scope operator

When an object goes out of scope in ParaSail it may require some amount of cleanup. For example, if the object provided access to some external resource, then when the object goes away, some release action on the external resource might be required. This is provided by the "end" operator. …

ParaSail module details …

ParaSail character, string, and numeric literals

… The *[five]* basic kinds of literals in ParaSail, and their corresponding universal types, are as follows:

kind of literal	example	universal type
string literal	"this is a string literal"	Univ_String
character literal	'c'	Univ_Character
integer literal	42	Univ_Integer
real literal	3.14159	Univ_Real
enum literal	#blue	Univ_Enumeration

The universal types can be used at run-time, but they are primarily intended for use with literals and in annotations. Univ_String corresponds to UTF-32, which is a sequence of 32-bit characters based on the ISO-10646/Unicode standard [4]. Univ_Character corresponds to a single 32-bit ISO-10646/Unicode character (actually, only 31 bits are used). Univ_Integer is an "infinite" precision signed integer type. Univ_Real is an "infinite" precision signed rational type, with signed zeroes and signed infinities. *[Univ_Enumeration is described in a later blog entry.]*

The universal numeric types have the normal four arithmetic operators, "+", "-", "*", "/". …

By providing conversions to and from a universal type, a "normal" type can support the use of the corresponding literal. These special conversion operations are declared as follows (these provide for integer literals):

```
operator "from_univ"(Univ : Univ_Integer)
  -> My_Integer_Type;
operator "to_univ"(Int : My_Integer_Type)
  -> Univ_Integer;
```

If an interface provides the operator "from_univ" converting from a given universal type to the type defined by the interface, then the corresponding literal is effectively overloaded on that type. The complementary operator "to_univ" is optional, but is useful in annotations to connect operations on a user-defined type back to the predefined operators on the universal types.

Annotations may be provided on the conversion operators to indicate the range of values that the conversion operators accept. So for a 32-bit integer type we might see the following:

```
interface Integer_32<> is
   operator "from_univ"
    (Univ : Univ_Integer {Univ in -2**31 .. +2**31-1})
     -> Integer_32;
   operator "to_univ"(Int : Integer_32)
     -> Result: Univ_Integer {Result in -2**31 .. +2**31-1};
   ...
end interface Integer_32;
```

With these annotations it would be an error to write an integer literal in a context expecting an Integer_32 if it were outside the specified range.

ParaSail Universal types in Annotations

As explained in the prior entry, ParaSail has *[five]* universal types, *[Univ_Enumeration]*, Univ_Integer, Univ_Real, Univ_Character, and Univ_String, with corresponding literals. The universal numeric types have the usual arithmetic operators, and the universal character and string types have the usual operations for concatenation and indexing. In some ways values of the universal types may be thought of as abstract mathematical objects, while the values of "normal" types are the typical concrete representations of such mathematical objects, with attendant limitations in range or precision. To connect a normal, concrete type to a corresponding universal type, the type defines the operator "from_univ" to convert from the universal type to the concrete type, and "to_univ" to convert back to the universal type.

In the prior entry, we learned that by defining "from_univ" a type can use the corresponding literal notation…. So what about the "to_univ" operator? If "to_univ" is provided, then ParaSail allows the use of a special convert-to-universal notation "[[...]]", which would typically be used in annotations, such as:

```
{ [[Left]] + [[Right]] in First .. Last }
```

This is equivalent to:

```
{ "to_univ"(Left) + "to_univ"(Right) in First .. Last }
```

This notation is intended to be a nod to the double bracket notation used in *denotational semantics* [5] to represent the *denotation* or abstract *meaning* of a program expression. Using this notation we can give a more complete definition of the semantics of the Integer interface:

```
interface Integer<First, Last : Univ_Integer> is
   operator "from_univ"
    (Univ : Univ_Integer {Univ in First .. Last})
     -> Integer;
   operator "to_univ"(Int : Integer)
     -> Result : Univ_Integer {Result in First .. Last};
   operator "+"(Left, Right : Integer
        {[[Left]] + [[Right]] in First .. Last})
     -> Result : Integer
        {[[Result]] == [[Left]] + [[Right]]};
   operator "-"(Left, Right : Integer
        {[[Left]] - [[Right]] in First .. Last})
     -> Result : Integer
        {[[Result]] == [[Left]] - [[Right]]};
   ...
end interface Integer;
```

Here we are defining the pre- and postconditions for the various operators by expressing them in terms of corresponding universal values, analogous to the way that denotational semantics defines the meaning of a program by defining a transformation from the concrete program notations to corresponding abstract mathematical objects.

January 2010

ParaSail thread versus task

In ParaSail, there is a pervasive, implicit parallelism. Because of that, it really doesn't make sense to talk about specific threads of control as there are potentially hundreds or thousands of them, appearing and disappearing all of the time. Hence any notion of thread identity is probably inappropriate. On the other hand, there might still be a reason to identify what might be called a *logical* thread of control, which would perhaps be reasonable to call a *task*, namely a logically separable unit of work. Here it might make sense to have a well defined task identity, and an ability to set a priority, or a deadline, or some sort of resource limits (such as maximum space, or maximum number of simultaneous threads, etc.). …

February 2010

Physical Units in ParaSail …

April 2010

ParaSail BNF …

May 2010

ParaSail enumeration types

We haven't talked about enumeration types in ParaSail yet. One challenge is how to define an enumeration type by using the normal syntax for instantiating a module, which is the normal way a type is defined in ParaSail. Generally languages resort to having special syntax for defining an enumeration type... But that seems somewhat unsatisfying, given that we can fit essentially all other kinds of type definitions into the model of instantiating a module. Another challenge with enumeration types is the representation of their literal values. In many languages, enumeration literals look like other names, and are considered equivalent to a named constant, or in some cases a parameterless function or constructor. Overloading of enumeration literals (that is using the same name for a literal in two different enumeration types declared in the same scope) may or may not be supported.

In ParaSail we propose the following model: We define a special syntax for enumerals (enumeration literals), of the form #name (e.g. #true, #false, #red, #green). We define a universal type for enumerals, Univ_Enumeration. We allow a type to provide conversion routines from/to Univ_Enumeration, identified by operator "from_univ" and operator "to_univ". If a type has a from_univ operator that converts from Univ_Enumeration, then that type is effectively an *enumeration* type, analogous to the notion that a type that has a from_univ that converts from Univ_Integer is essentially an *integer* type. As with other types, double-bracket notation applied to a value of an enumeration type, e.g. [[enumval]], is equivalent to a call on to_univ(enumval), and produces a result of Univ_Enumeration type. An additional special Boolean "in" operator which takes one Univ_Enumeration parameter determines which enumerals are values of the type. The notation "X in T" is equivalent to a call on the operator T::"in"(X). The precondition on from_univ would generally be {univ in Enum_Type}.

Here is an example:

```
interface Enum<Enumerals :
   Vector<Univ_Enumeration>> is
  operator "in"(Univ_Enumeration) -> Boolean<>;
  operator "from_univ"
   (Univ : Univ_Enumeration {Univ in Enum}) -> Enum;
  operator "to_univ"(Val : Enum)
   -> Result: Univ_Enumeration { Result in Enum };
  ...
end interface Enum;
...
type Color is Enum<[#red,#green,#blue]>;
var X : Color := #red;  // implicit call on from_univ
```

Here we presume the class associated with the Enum interface implements "in", "from_univ", and "to_univ" by using the Enumerals vector to create a mapping from Univ_Enumeration to the appropriate value of the Enum type, presuming the enumerals map to sequential integers starting at zero. A more complex interface for creating enumeration types might take such a mapping directly, allowing the associated values to be other than the sequential integers starting at zero. The built-in type Boolean is presumed to be essentially Enum<[#false,#true]>.

So why this model? One is that it fits nicely with the way that other types with literals are defined in ParaSail, by using from_univ/to_univ. Secondly, it naturally allows overloading of enumeration literals, in the same way that the types of numeric literals are determined by context. Finally, it makes a strong lexical distinction between enumeration literals and other names. This could be considered a bug, but on balance we believe it is useful for literals to stand out from, and not conflict with, normal names of objects, types, operations, and modules. Note that this model is reminiscent of what is done in Lisp with the quote operator applied to names, e.g. 'foo. It also borrows from the notation in Scheme of #f and #t for the literals representing false and true.

ParaSail without pointers?

We are working on defining the semantics for pointers in ParaSail. As described in an earlier entry, we plan to use region-based storage management rather than more fine-grained garbage collection. However, there is the question of how pointers are associated with their region, and more generally how pointers are handled in the type system. We are considering a somewhat radical approach: effectively *eliminate* most pointers, replacing them with *generalized container indexing* and *expandable objects*. …

By *generalized container indexing*, we mean using an abstract notion of index into an abstract notion of container. The most basic kind of container is an array, with the index being an integer value or an enumeration value. Another kind of container is a hash table, with the index (the key) being something like a string value. …

By *expandable objects*, we mean treating a declared but not initialized object (including a component of another object) as a kind of stub into which a value can be stored. Even an initialized object could be overwritten via an assignment which might change the size of the object. Finally, an initialized object could be set back to its stub state, where the object doesn't really exist anymore….

Expandable objects eliminate a common use of pointers as a level of indirection to support objects of variable or unknown size. We propose to provide qualifiers **optional** and **mutable** on objects to indicate that they are expandable. The qualifier **optional** means that the object starts out **null**, but may be assigned a non-**null** value. The qualifier **mutable** means that the object may be assigned multiple values of different sizes during its lifetime. If both are given, then the object starts out **null**, can be assigned non-**null** values of varying size during its lifetime, and can be assigned back to **null**. …

Handling concurrent events in ParaSail …

Talking about ParaSail …

June 2010

Generalized For loops in ParaSail

As we start to dive into the lower-level control structures for ParaSail, one clearly very important structure is the **for** loop. We have already suggested the basic structure:

```
for I in 1..10 [forward | reverse | concurrent] loop
```

However, it would be nice if the **for** loop could also be used for more general looping structures. A number of languages have adopted a for loop where there is an initialization, a "next step", and a termination/continuation condition….

The above considerations lead us to the following possible approach to a generalized for loop in ParaSail…:

```
for T => Root then T.Left || T.Right
  while T != null loop
    … loop body
end loop
```

This would represent a "loop" which on each iteration splits into two threads, one processing T.Left and the other T.Right. The iteration stops when all of the threads have hit a test for T != null which returns false. …

If we were to specify **concurrent loop** then it would presumably imply that the daughter threads are to be created immediately once the test was found to be true, not waiting for the loop body of the parent thread to finish. This would effectively create a thread for every node in the tree, with the loop body executing concurrently for each node.

An intentional race condition in a ParaSail concurrent loop

An intriguing question, given the presence of the concurrent loop construct in ParaSail, is what would happen if there were a loop **exit** or a **return** statement in the body of a **concurrent** loop. Earlier we have said that only **forward** and **reverse** loops would allow an **exit** statement. But perhaps we should allow an **exit** or a **return** from within a concurrent loop as a way of terminating a concurrent computation, aborting all threads besides the one exiting or returning. This would essentially be an intentional race condition, where the first thread to **exit** or **return** wins the race. This would seem to be a relatively common paradigm when doing a parallel search, where you want to stop as soon as the item of interest is found in the data structure being walked, for example, and there is no need for the other threads to continue working.

…

For example, using the generalized for loop from the previous posting:

```
var Winner : optional Node := null;
for T => Root then T.Left || T.Right
  while T != null concurrent loop
    if T.Key == Desired_Key then
       exit loop with Winner => T;
    end if;
end loop;
```

At this point Winner is either still **null** if no node in the tree has the Desired_Key, or is equal to some node whose Key equals the Desired_Key. If there is more than one such node, it would be non-deterministic which such node Winner designates.

Clearly this kind of brute force search of the tree would not be needed if the tree were organized based on Key values. This example presumes the Key is not controlling the structure of the tree. Also note that if this were a function whose whole purpose was to find the node of interest, the **exit loop with** ... statement could presumably be replaced by simply **return** T and we wouldn't need the local variable Winner at all.

Flex-based lexical scanner for ParaSail …

(A)Yacc-based parser for ParaSail …

Some additional examples of ParaSail …

July 2010

N Queens Problem in ParaSail

Here is a (parallel) solution to the "N Queens" problem in ParaSail, where we try to place N queens on an NxN chess board such that none of them can take each other. This takes the idea of using the "**continue**" statement as a kind of implicit recursion to its natural conclusion. This presumes you can turn a "normal" data structure like Vector<> into a concurrent data structure by using the keyword "**concurrent**," which presumably means that locking is used on all operations to support concurrency. It is debatable whether this use of a "**continue**" statement to effectively start the next iteration of a loop almost like a recursive call is easier or harder to understand than true recursion. *[This example has been updated and corrected.]*

```
interface N_Queens <N : Univ_Integer := 8> is
   // Place N queens on an NxN checkerboard so that
   // none of them can "take" each other.
   type Chess_Unit is new Integer<-N*2 .. N*2>;
   type Row is Chess_Unit {Row in 1..N};
   type Column is Chess_Unit {Column in 1..N};
   type Solution is Array<optional Column, Indexed_By => Row>;

   function Place_Queens() -> Vector<Solution>
    {for all Sol of Place_Queens => for all Col of Sol => Col not null};
end interface N_Queens;

class N_Queens is
   type Sum_Range is Chess_Unit {Sum_Range in 2..2*N};
   type Diff_Range is Chess_Unit {Diff_Range in (1-N) .. (N-1)};
   type Sum is Set<Sum_Range>;
   type Diff is Set<Diff_Range>;
 exports
```

```
  function Place_Queens() -> Vector<Solution>
   {for all Sol of Place_Queens => for all Col of Sol => Col not null}
  is
   var Solutions : concurrent Vector<Solution> := [];
  *Outer_Loop*
   for (C : Column := 1; Trial : Solution := [.. => null];
    Diag_Sum : Sum := []; Diag_Diff : Diff := []) loop
     // Iterate over the columns
     for R in Row concurrent loop
       // Iterate over the rows
       if Trial[R] is null and then
        (R+C) not in Diag_Sum and then
        (R-C) not in Diag_Diff then
         // Found a Row/Column combination that is
         // not on any diagonal already occupied.
         if C < N then
           // Keep going since haven't reached Nth column.
           continue loop Outer_Loop with (C => C+1,
            Trial     => Trial | [R => C],
            Diag_Sum  => Diag_Sum | (R+C),
            Diag_Diff => Diag_Diff | (R-C));
         else
           // All done, remember result with last queen placed
           Solutions |= (Trial | [R => C]);
         end if;
       end if;
     end loop;
   end loop Outer_Loop;
   return Solutions;
  end function Place_Queens;
end class N_Queens;
```

Updated (A)Yacc grammar for ParaSail …

Pointer-free primitives for ParaSail …

August 2010

Eliminating the need for the Visitor Pattern in ParaSail …

Ad hoc interface matching in ParaSail …

Initial implementation model for ParaSail types …

No exceptions in ParaSail, but exitable multi-thread constructs

We have been mulling over the idea of exceptions in ParaSail, and have pretty firmly concluded that they aren't worth the trouble. In a highly parallel language, with lots of threads, exception propagation across threads becomes a significant issue, and that is a nasty area in general. Also, exceptions can introduce many additional paths into a program, making thorough testing that much harder. And the whole business of declaring what exceptions might be propagated, and then deciding what to do if some other exception is propagated can create numerous maintenance headaches.

There is a feature in ParaSail as currently designed which provides some of the same capabilities of exceptions, but is particularly suited to parallel programming. This is the "**exit with**" statement, which allows a construct to be exited with one or more values specified as results, and at the same time terminating any other threads currently executing within the construct. For example, here is a loop implementing a parallel search of a tree with the first thread finding the desired node exiting and killing off all of the other threads as part of the "**exit ... with**" statement:

```
const Result : optional Tree_Id;
for T => Root then T.Left || T.Right
  while T not null concurrent loop
    if T.Value == Desired_Value then
       // Found desired node, exit with its identifier
       exit loop with (Result => T.Id);
    end if;
end loop with (Result => null);
```

This declares a Result object of type Tree_Id. It then walks the tree in parallel, starting at Root and continuing with T.Left and T.Right concurrently. It continues until it reaches "null" on each branch, or some node is found with its Value component matching the Desired_Value. The value of Identifier at the end indicates the identifier of the node having the desired Value, or null to indicate that no node was found. The presence of **optional** in the declaration for Result indicates that its value might be null.

Supporting this kind of intentional "race" seems important in parallel programming, as many problems are amenable to a divide and conquer approach, but it is important that as soon as a solution is found, no further time is wasted searching other parts of the solution space. The "**end ... with**" phrase allows the specification of one or more results if the construct ends normally, as opposed to via an "**exit ... with**" (in this case, ending normally means all threads reach a null branch in the walk of the tree without finding the desired node). Effectively the "**exit ... with**" skips over the "**end ... with**" phrase.

So how does this all relate to exceptions? Well given the "**exit ... with**" capability, one can establish two or more threads, one which monitors for a failure condition, and the others which do the needed computation. The thread monitoring for a failure condition performs an "**exit ... with**" if it detects a failure, with the result indicating the nature of the failure, and as a side-effect killing off any remaining computation threads. If the normal computation succeeds, then an "**exit ... with**" giving the final result will kill off the monitoring thread. Note that the "**exit ... with**" statements must occur textually within the construct being exited, so it is visible whether such a premature exit can occur, unlike an exception which can arise deep within a call tree and be propagated out many levels.

As an example of the kind of failure condition which might be amenable to this kind of monitoring, imagine a resource manager object, which provides up to some fixed maximum of some kind of resource (e.g. storage) to code within a block. This resource manager (which is presumably of a concurrent type) could be passed down to operations called within the block for their use. Meanwhile, a separate monitoring thread would be created immediately within the block which would call an operation on the resource manager which would suspend the thread until the resource runs out, at which point it would be awakened with an appropriate indication of the resource exhaustion, and any other information that might be helpful in later diagnosis. On return from this Wait_For_Exhaustion operation, the monitoring thread would do an "**exit block with** (Result => Failure, ...)" or equivalent, to indicate that the computation required more resources than were provided. The code following the block would then be able to take appropriate action.

September 2010 …

[Please continue reading the blog on the web if interested. We hope the above extract has captured the essence of the ParaSail language design process.]

Conclusions about ParaSail

So what have we accomplished? The design of ParaSail is now largely complete. As of this writing, a grammar exists and a parser based on it is working; a simplified ParaSail Virtual Machine interpreter has been implemented, and a prototype compiler is under develoment. ParaSail has been presented at several conferences and other venues. So far it has been quite well received. In June at the upcoming AdaEurope Ada Connection conference in Edinburgh, there will be a workshop/tutorial allowing experimentation with the language and its prototype implementation. But of course, a programming language is only of significant value if it is *used* to build software.

Our view is that to succeed in the “multicore” world, we need new languages that make parallel programming as productive as sequential programming, without adding further complexity to the already challenging job of writing correct and secure software. By making parallel expression evaluation the default, by making it easy to insert even more race-free parallelism explicitly, and by integrating compile-time checked preconditions, postconditions, and various other kinds of assertions into a unified language from day one, we believe ParaSail can be one of those new languages that carry us into the multicore era.

References

[1] S. T. Taft (2011), *Designing ParaSail, a new programming language*, http://parasail-programming-language.blogspot.com .

[2] Stanford Artificial Intelligence Laboratory (1976), *SAIL*, http://pdp-10.trailing-edge.com/decuslib20-01/01/decus/20-0002/sail.man.html .

[3] R. E. Strom, *et al* (1991), *Hermes: A Language for Distributed Computing*, Prentice-Hall, Series in Innovative Technology, ISBN 0-13-389537-8.

[4] Unicode Consortium (2011), *Unicode 6.0.0*, http://www.unicode.org/versions/Unicode6.0.0/.

[5] L. Allison (1987), *A Practical Introduction to Denotational Semantics,* Cambridge University Press.