AdaCore

Build Software that Matters

ParaSail: *A Pointer-Free Pervasively-Parallel Language for Irregular Computations*

S. Tucker Taft, AdaCore

<Programming> 2019, Genoa, Italy

April 2019

Goals for **ParaSail**
Parallel Specification and implementation language

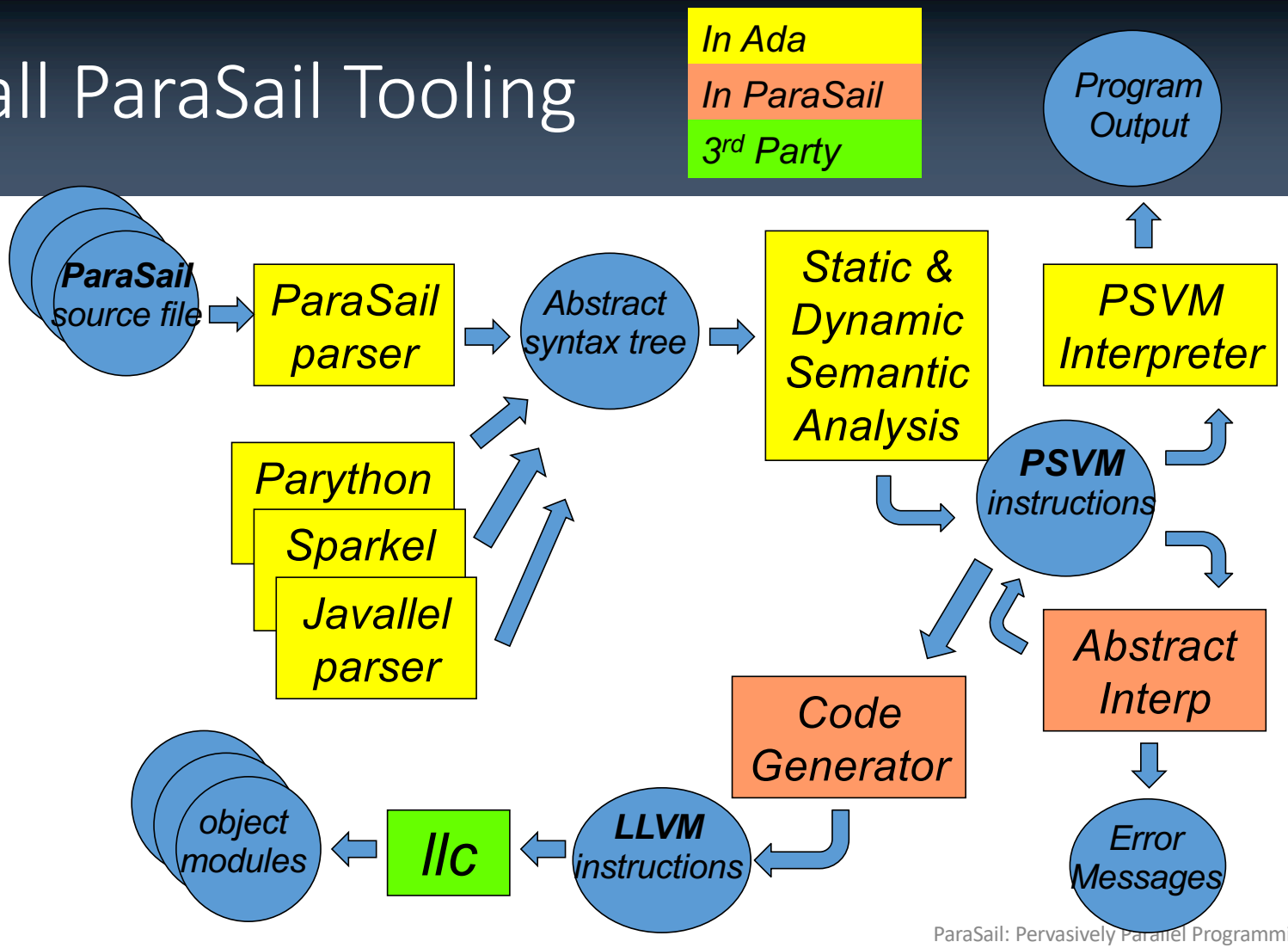
- Goal A: Simplify writing parallel programs
  - Make it as easy for programmer to write in parallel as it is sequentially
  - Make it easy for compiler to detect data races
  - Make it easy for compiler to insert parallelism

- Goal B: Simplify writing safe, correct, and predictable code
  - Provide automatic storage management without garbage collection
  - Integrate contract-based programming annotations into language
  - Eliminate difficult-to-verify features

History of ParaSail

- **2009 – Started design of ParaSail, on a blog**
  - My friends were too busy – drowned my sorrows in a ***blog***
  - Design documented along the way on *http://parasail-programming-language.blogspot.com*
- **2011 – ParaSail interpreter completed**
- **2013 – Refactored to support multiple parsers, same AST**
  - ***Javallel, Parython, Sparkel*** *– to address* ***surface syntax*** *preferences*
- **2014 – LLVM-based code generator built**
  - Parallelism is only interesting if it makes the program ***faster***
  - Written in interpreted ***ParaSail*** using *reflection (by a summer intern)*
- **2015 – Abstract-interpretation-based static analysis tool built:**
  - ***ParaScope***: **Para**Sail **S**tatic **C**atcher **o**f **P**rogramming **E**rrors
  - Written in interpreted ***ParaSail*** using *reflection*
- **2017-8 – Refactored LLVM-based code generator**
  - Use ***register*** model rather than ***stack*** model for temps and parameters in LLVM
  - ***Inline*** more of the run-time support

Overall ParaSail Tooling



Start with Goal B: Simplify and Unify Language to "make room" for Parallelism and Contracts

- Modules and Types – Can we simplify and unify?
  - Package, Template, Namespace, Class, Interface, Object, Record, Struct, Structure, Sig, Signature, Monitor, Task
    - Do we really need more than one of these?
    - ***What unifies these?***
  - ➢ Their components are heterogeneous at compile-time
  - ➢ A symbolic selector identifies a component and determines its type.
  - ➢ Some are parameterized, some are not.
  - ➢ Some are modules, some are types, some are both.
  - ➢ Some define an interface; some define an implementation with an implicit interface
- What about Operations – must we have so many kinds?
  - Virtual function, Static function, Constructor, Destructor, Method, Procedure
    - Is it Object.Operation(...) vs. Operation(Object, ...)?
    - ***What unifies these?***
  - ➢ Algorithm with inputs, outputs, and local variables, and possibly up-level access to globals
  - ➢ Inheritance of interface and optionally implementation across type hierarchy
    - Indirect binding as part of interface inheritance
- Objects – value vs. reference model? mutable vs. immutable?

ParaSail "Simplified" Building Blocks

- **Module** *– always parameterized, with separate interface*
  - *Parameterized* unit with *Interface* part, and optionally an *Implementation* part
  - **interface** Enum <Literals : Vector <Univ_Enumeration>> **is** ...
  - Can *extend* another module (code and data are inherited along with interface)
  - Can *implement* one or more module's interfaces (only interface is inherited)
- **Type** *– single syntax for all type declarations*
  - An *instance* of a Module: **type** T **is** [**new**] M < . . . > *// "**new**" means use "name" equivalence*
  - e.g. **type** Color **is new** Enum <[#red, #green, #blue]>
  - "T+" is *polymorphic* type representing any type implementing T's interface
- **Object** *– only "value" types, **var** or **const***
  - An *instance* of a Type (type can be inferred from initial value)
    - has an updatable *value* if declared "**var**"; can be **null** if declared "**optional**"
  - **var** Obj : T := Create(...) *// ParaSail allows overloading on parameter and result types*
- **Operation** *– role of operation determined by parameters and where declared*
  - *Defined* in a Module, and
  - *Operates* on one or more Objects of specified Types
  - Only operates on its *explicit* parameters; can *update* only those declared "**var**"

A Map Module Interface in ParaSail

```
interface PSL::Containers::Map          // A hashed-map module; a set of K/V Pairs
   <Key_Type is Hashable<>; Value_Type is Assignable<>>   // formal params of module
is
   op "[]"() -> Map                     // Return an empty map
   type Pair is Key_Value<Key_Type, Value_Type> // K/V pair -- actual params in <...>
   op "|="(var M: Map; KV: Pair)        // Add [Key=>Value] pair to Map
   op "|"(M: Map; KV: Pair) -> Map      // Return Map with [Key=>Value] pair added
   op "index_set"(M: Map) -> Set<Key_Type>      // Return set of keys of the map
   op "in"(Key: Key_Type; M: Map) -> Boolean    // True if Key in "index_set"(M)
   op "-="(var M: Map; Key: Key_Type)           // Remove mapping for Key, if any
   op "indexing"(ref M: Map; Key: Key_Type) {Key in M} -> ref Value_Type
         // Used for indexing via "M[Key]"; {...} is precondition if before "->"
   func Remove_Any(var M: Map) -> optional Pair
         // Remove one mapping from Map; Return null if map is empty.
   op "magnitude"(M: Map) -> Univ_Integer
         // Number of mappings in the table; Supports "|M|" notation
   func Is_Empty(M: Map) -> Boolean     // True if no mappings in M
end interface PSL::Containers::Map      // end bracketing is optional if idented properly
```

Key to listing:
func: callable function/method/operation
op: function that relies on *syntactic sugar*
var: allows update of parameter
ref: short-lived reference to existing object
type: nested type; instance of module
{…}: contract (pre/postcondition, invariant, …)

Example uses of Map Module

var X : Map <Key_Type => String, Value_Type => Integer> := []	*mutable map, string => integer*
const Y : Map <String, Integer> := [“the” => 35, “cat” => 42]	*immutable map; uses “[]” & “|=“*
X |= [“dog” => 2]	*add key/val pair to X*
var Z := Y | X	*type inferred from initial value*
type String_Int_Map **is** Map<String, Value_Type => Integer>	*named type, but equiv to anon*
type Word_Counts **is new** Map<String, Integer>	*named type, **not** equiv to anon*
var WCs : Word_Counts := Z *// illegal – type mismatch*	*“**new**” forces **name equivalence***
for each [W => C] **of** Z **loop** *// could be **{forward, reverse, concurrent} loop*** IO.Stdout.Println(“Word: `(W), Count: `(C)”) **end loop** *// **end** bracketing is optional*	***syntactic sugar** uses operations of Map (“index_set”, “indexing”) and Set (Remove_Any [or Remove_First, Remove_Last])* *`(…) is string **interpolation** syntax*

Parts of Map Module Implementation

```
class Map is
   interface Node<> is   // nested local module
      var Entry: Pair
      var Next: optional Node
   end interface Node
   var Backbone: Basic_Array<optional Node>
   var Count: Univ_Integer := 0
exports  // "exports" separates internal part from bodies of exported operations
   . . .
   op "index_set"(M : Map) -> Result : Set<Key_Type> is  // Return set of keys of the map
      Result := []
      for each B of T.Backbone loop
         for N => B then N.Next while N not null loop
            Result |= Key_Of (N.Entry)
         end loop
      end loop
   end op "index_set"
   . . .
end class Map  // end bracketing optional
```

internal part

exported part

Aside: How to live without pointers? Use *Expandable* Objects Instead

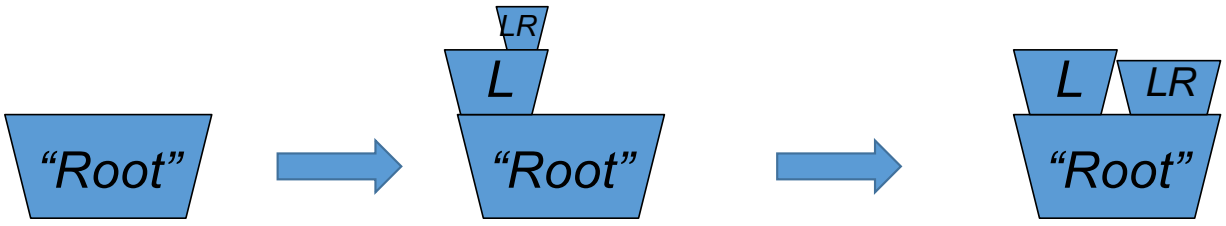
- All types have additional **null** value, which takes very little space
  - Only objects or components declared **optional** can be null; allows object to grow and shrink
  - Eliminates many of the common uses for pointers, e.g. trees
  - Assignment ("**:=**") is by copy
    - Move ("**<==**") and swap ("**<=>**") operators also provided
  - Essentially equivalent to an implicit "ownership" model (cf. Rust, unique_ptr, etc.)
- Generalized indexing into objects replaces pointers for cyclic graph structures
  - **for each** N **of** Directed_Graph[I].Successors **loop** ...
- Region-Based Storage Mgmt can replace Global Heap
  - All objects are "local" with growth/shrinkage using local heap
  - "**null**" value carries indication of region to use on growth
- Short-lived references to existing objects are permitted
  - Returned by user-defined indexing functions, for example
  - Useful for defining "slices" of an array or other container

Example: Pointer-Free Trees

```
interface Tree_Node <Contents_Type is Assignable<>> is
   var Contents: Contents_Type
   var Left : optional Tree_Node := null
   var Right : optional Tree_Node := null
end interface Tree_Node

var Root : Tree_Node<Univ_String> := (Contents => "Root")
Root.Left := (Contents => "L", Right => (Contents => "LR"))
Root.Right <== Root.Left.Right  // Root.Left.Right now null
```

The Flower-Pot model of objects



Example: Pointer-Free Linked List

```
interface Linked_List<Elem_Type> is
    var Data : Elem_Type
    var Next : optional Linked_List := null
end interface Linked_List

LL : Linked_List <String> :=
  (Data => “Hello”, Next => (Data => “World”))

LL.Next.Data := “Universe”
```

And now Goal A: Pervasive Parallel Programming

- Make parallel evaluation the default
  - F(X) + G(Y)
    - Goal: safely evaluate F(X) and G(Y) in parallel without looking inside F & G
    - Even if F has side effects on X, or G has side effects on Y
  - Seemingly trivial with pure functional programming language
    - But would disallow any update-in-place optimizations
    - Monads can re-create their own parallelization challenges
    - Any “breaking” of purity as in ML family “ref”s brings back all of the challenges
  - Hard if X and Y use unrestricted pointers, since both might point to “Z”
  - Hard if F and G can make unrestricted use of globals w/o explicit synchronization
    - But any synchronization would be bad for performance

Goals A & B: Eliminate features that get in the way

- Eliminate global variables
  - Operation can only access or update variable state via its parameters
- Eliminate parameter aliasing
  - Use “hand-off” semantics (cf. Hermes language)
- Eliminate explicit threads, lock/unlock, signal/wait
  - Concurrent objects synchronized automatically
- Eliminate run-time exception handling
  - Can use compile-time checking and propagation of preconditions
  - “Mutually Assured Destruction” of sibling threads on exit is supported as an alternative.
- Eliminate pointers
  - Adopt notion of “optional” objects that can grow and shrink
  - User-defined indexing as alternative for cyclic graph structures
- Eliminate global heap and garbage collector
  - Replaced by region-based storage management (local heaps)
  - All objects conceptually live in a local stack frame

Example: An Implicitly Parallel Divide-and-Conquer Recursive Not-In-Place ("functional") QSort

```
func QSort(V : Vec_Type is Vector<Comparable<>>) -> Vec_Type is
   if |V| <= 1 then
      return V                          // The easy case
   else
      const Mid := V[ |V|/2 ]    // Pick a pivot value
      return  //  implicit parallelism inserted here on recursive calls
         QSort( [for each E of V {E < Mid} => E] )  // Recurse; {. . .} is filter
       |    [for each E of V {E == Mid} => E]          // No recursion since all values equal the pivot
       | QSort( [for each E of V {E > Mid} => E] )  // Recurse
   end if
end func QSort
```

Number of Cores	Core Configuration	Wall Clock	CPU	Thread Utilization	Speedup
1	Single Threaded	77.0	76.0	98.6	1 x
2 x 2	Dual Core x 2 Hyper Threads/Core	42.6	142.2	333.8	1.8 x

An Explicitly Parallel In-Place Non-Recursive Qsort
(see paper for additional explication)

```
func Quicksort(var A : Array_Type) is  // sort array A in place
   for Arr => A[..] while |Arr| > 1 loop // process slices
      if |Arr| == 2 then // short slice: simple case
         if Arr[Arr.Last] < Arr[Arr.First] then
            Arr[Arr.First] <=> Arr[Arr.Last]  // swap
      else                    // long slice: partition array
         const Mid := Arr[Arr.First + |Arr|/2]
         var Left := Arr.First
         var Right := Arr.Last
*Swapping* until Left > Right loop  // keep swapping
            var New_Left := Right+1
            var New_Right := Left-1
          then // "then" waits for prior processing
            for I in Left .. Right forward loop // scan from left
               if Mid <= Arr[I] then
                  New_Left := I // Item OK for right partitition
                  if Mid < Arr[I] then
                     exit loop  // Item must go in right partition
          ||    // "||" requests concurrent processing
            for J in Left .. Right reverse loop // scan from right
               if Arr[j] <= Mid < Arr[J] then
                  New_Right := J   // Item OK for left partition
                  if Arr[J] < Mid then
                     exit loop  // Item must go in left partition
          then //  waits for the above "for I ... || for J ..." to complete
            if New_Left > New_Right then  // check if any more swaps
               Left := New_Left
               Right := New_Right
               exit loop Swapping  // no more swapping needed
            Arr[New_Left] <=> Arr[New_Right]  // swap items
            Left := New_Left + 1  //  skip past swapped items
            Right := New_Right - 1
         end loop Swapping  // continue looking for items to swap
      then  //  add new slices to "bag"  of slices to sort in parallel
         continue loop with Arr => Arr[Arr.First .. Right]
      ||  // "||" is higher precedence than "then"
         continue loop with Arr => Arr[Left .. Arr.Last]
```

An interesting Parallel programming idiom

Concurrent loop(s) plus explicit "**continue loop with** X => …"
Replaces recursion with parallel worklist-like idiom

Example 1: Qsort of A:

```
for Arr => A[..] while |Arr| > 1 loop  // start with full-array "slice" A[..]
   … partition slice "Arr" using swapping, and process two sub-slices
   //  use "continue loop" to add work items to the "bag" of iterations
   continue loop with Arr => Arr[Arr.First .. Right]
|| // "||" is pronounced "in parallel with …"
   continue loop with Arr => Arr[Left .. Arr.Last]
end loop  // wait for "bag" of iterations to complete
```

“Bag of Iterations” Idiom, Example 2: N-Queens

```
const Rows : Countable_Range<Chess_Unit> := 1..N
var Solutions : concurrent Vector<Board_State> := []   // “concurrent” means locking

*Outer_Loop*
 for Partial : Board_State := No_Columns(N) loop      // Start with no columns filled
   for R in Rows concurrent loop                      // Try all rows in next column
      const Next := Add_Queen(Partial, At_Row => R) // Add queen at given row
      if Is_Safe(Next) then                          // See if is a safe place for queen
         if Num_Columns(Partial) == N then     // New queen is OK; see if done
           Solutions |= Next                   // Remember another complete solution
         else
           continue loop Outer_Loop
              with Partial => Next     // Add partial solution to bag of (Outer) work items
         end if
      end if
   end loop
 end loop Outer_Loop
```

Other **ParaSail** Tidbits

- Compile-time computation thanks to full-language interpreter
  - Can use types produced by module instantiations
  - Literals, any self-contained computation, computed at compile-time
    - Can use assertions/pre-conditions for “type checking”
- Heavy use of Syntactic sugar; Syntax agnostic semantics
  - Simple unified underlying model for defining abstractions
    - No “implicit” parameter in decl, but allow Obj.Operation(…) notation
  - Compact/familiar/expressive syntax for use of abstractions by client
  - Supports multiple surface syntaxes
- Safe Distributed and Concurrent Programming
  - pointer-free (logical) objects are “self contained”
- Contract Annotations in the language
  - recognized by both the interpreter and the compiler
  - compile-time proved, run-time checked, or hybrid

Evaluation of ParaSail against its Goals

- *Use syntax and type model familiar to existing professional programmers*
  - Anecdotal evidence for readability and understandability from existing C++, Java, Python programmers
    - Though a bit verbose for some!
  - Support for alternative surface syntax is of interest to some
- *Ease the creation of safe, parallel programs of significant size*
  - LLVM code generator written by parallel programming neophyte during 6-month internship
  - High levels of parallelism achieved without significant extra effort
    - Lack of globals and compile-time race detection biggest help
- *Raise the level of abstraction*
  - Static analysis tool based on abstract interpretation re-implemented in ParaSail
  - Value Propagation algorithm significantly simpler despite lack of pointers and globals
    - Largely thanks to pointer-free data structuring and concise syntactic sugar available to all user-defined data types
- *Support real-time embedded parallel programming*
  - Support for both compile-time and run-time synchronization
  - Bounded, predictable storage management
- *Achieve these goals with economy of means*
  - Simple Unified Module/Type/Object/Operation Building Blocks
  - "50-page" reference manual a la Modula-3

Next Steps for ParaSail

- *Performance, performance, performance*
  - Inlining more of the run-time support
  - Stack-resident (non-**optional** or bounded) compound objects
  - Inlining non-**optional** component objects into enclosing object
  - Eliminating header overhead for non-**optional** component objects
- *Parython*
  - Productize Python variant of ParaSail
  - Parallel, type-checked, interpreted or compiled-to-LLVM Python
- *Take more advantage of full interpreter at compile-time*
  - Currently used for checking properties of literals
    - Enum Literal is part of enumeration, Integer literal is in range, String literal has desired form
  - Any ParaSail expression with no use of variables is evaluated at compile time
    - Could formalize into a separate "user-directed type-check" step
      - cf. Ruby **comp-types** by Jeff Foster (Tufts/UMD)

Conclusion : Overall **ParaSail** Themes

- *Mutable Objects* with Value Semantics
- *Stack-Based* Heap Management
- *Compile-Time* Exception Handling
- *Race-Free* Parallel Programming

Expressivity of functional programming, but with safe mutation, easy parallelization, and ability to model the "real" world

Downloadable from: parasail-lang.org

Forum: groups.google.com/group/parasail-programming-language/

Contact: taft@adacore.com

Blog: http://parasail-programming-language.blogspot.com/