Designing a Parallel Programming Language while Blogging

S. Tucker Taft
AdaCore
December, 2019

First Blog Entry -- September 25, 2009

Designing ParaSail, a new programming language

This blog will follow 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 tentatively named "ParaSail" for Parallel, Specification and Implementation Language.

Friday, September 25, 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

Logo for ParaSail

Second Blog Entry -- September 26, 2009

Saturday, September 26, 2009

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.

So what is it like to design while blogging?

- Blogging is basically a “solo” process, but with the possibility of feedback
- A blog is somewhere between a stream of consciousness, a brainstorming session, and an after-the-fact rationalization
- Explaining ideas helps to sharpen them, or to expose their weaknesses, or both
- No hard “team” deadlines means hard problems can be postponed, allowing them hopefully to soften up over time
- Eventually a solution may present itself, after enough ***groundwork*** *and* ***context*** have been provided
- *The design process can begin to feel more like* ***discovery*** *than* ***invention***

Groundwork and Context:

start with your goals

Third Blog Entry -- September 26, 2009

ParaSail language themes and philosophy

So what will make **ParaSail** an interesting programming language? What is the philosophy behind the language? … 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. … 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 ...

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

Establish some basic principles

Simplify, simplify, simplify

- 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

Early ideas

- Eliminate run-time exception handling
  - Can use compile-time checking and propagation of preconditions
- 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

Later ideas

Do some initial design

ParaSail "Simplified" Building Blocks

- **Module** – *always parameterized, with separate interface*
  - *Parameterized* unit with *Interface* part, and optionally an *Implementation* part
  - **interface** Ordered_Set <Elem_Type is Comparable> **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 data type declarations*
  - An *instance* of a Module: **type** T **is** [**new**] M < . . . > *// "**new**" means use "name" equivalence*
  - e.g. **type** String_Set **is new** Ordered_Set <String>
  - "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**"

Focus in on how types are defined

- **Type** – *single syntax for all data type declarations*
  - An *instance* of a Module: **type** T **is** [**new**] M < . . . >
    *// "**new**" means use "name" equivalence*
  - e.g.:

```
type Percent  is new Integer <0 .. 100>
type Phone_Book  is new
    Map <Key => String, Value => Phone_Number>
type String_Set  is Ordered_Set <String>
```

Bump into some road blocks:

Enumeration types

How do enumeration types fit into this syntax?

- Jensen and Wirth popularized enumeration types in Pascal
- Wirth then dropped enumeration types from Oberon
  - *“Enumeration types appear to be a simple enough feature to be uncontroversial. However, they defy extensibility over module boundaries. Either a facility to extend given enumeration types has to be introduced, or they have to be dropped. …” (From Modula to Oberon, 1988)*
- Bertrand Meyer did not include them in Eiffel
  - *“Introducing Pascal-ilke enumeration types would be a conceptual disaster in Eiffel: they would conflict with the type system of the language, which is otherwise simple (the four simple types on the one hand, and the class types on the other). It does not seem feasible to combine this notion elegantly with inheritance.” (Object-Oriented S/W Construction, 1988)*
- James Gosling did not include them in original Java
  - *“Enumerations were left out of the Java spec not because I think they're a bad idea, but because I couldn't converge on a design that made sense. …” (JavaWorld, June 1, 1998)*

What is so hard about enumeration types?

- The enumeration literals
  - They do not seem to work like numeric or string literals
  - Each type has its own set of literals, which are defined as a side effect of defining the type
- Where do the literals “live” in the program’s lexical scopes?
- Do they allow overloading -- can two enumeration types have the same literals?
- When you “import” a type do you automatically import all of its literals?
- Can an extension add more enumeration literals? How does that work?
- In ParaSail: How can you fit the need to define a bunch of new enumeration literals into the syntax of an instantiation of some pre-existing module?

How do numeric and string literals work?

- They already exist before the declaration of any types
- Something about the type determines what sort of literals you can use with it
  - An integral type can use integer literals
  - A floating point type can use floating point literals
  - A string type can use string literals
- In ParaSail
  - If a type has “from_univ” routine from a “Univ” type it can use the corresponding literals

Universal Type	**Syntax of Literals**
Univ_Integer	42
Univ_Real	3.141592653589793
Univ_String	“Hello, world!”
Univ_Character	‘@’

Example of Integer module in ParaSail

```
interface Integer <Range : Interval<Univ_Integer>> is
   op “+” (Left, Right : Integer) -> Integer
   op “-” (Left, Right : Integer) -> Integer
   ...
   op “from_univ” (Univ: Univ_Integer) {Univ in Range} -> Integer
                        //  Can use Univ_Integer literals that satisfy precondition
   op “to_univ” (Integer) -> Result : Univ_Integer {Result in Range}
                        // Reverse conversion for output, result satisfies postcondition
  ...
end interface Integer
```

Can we fit enum types into this pattern?

Yes, just have to let it gestate for nine months

...

Blog entry -- May 15, 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. …

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 **op** "`from_univ`" and **op** "`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. ...

Basic trick:

pre-create a
Univ_Enumeration type
with all enumeration
literals imaginable,
analogous to
Univ_Integer

An Enum module to define enumeration types

```
interface Enum<Enumerals : Vector<Univ_Enumeration>> is
  op "from_univ" (Univ : Univ_Enumeration) {for some E of Enumerals => Univ == E} -> Enum
        //  can use any literal of type Univ_Enumeration that satisfies the precondition
  op "to_univ"(Val : Enum) -> Result: Univ_Enumeration {for some E of Enumerals => Result == E}
        //  reverse conversion for output will satisfy postcondition
  ...
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 **`"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]>`**.

Defining an enum type with a specified “rep”

```
interface PSL::Core::Enum_With_Rep
   <Rep_Type is Imageable<>;  Rep_Map : Two_Way_Map<Univ_Enumeration, Rep_Type>> is
        // An enumeration type specified using a map from literal to value
        // of an underlying representation type.
        op "from_univ"(Univ : Univ_Enumeration)
           {for some [Lit => Val] of Rep_Map => Lit == Univ}
             -> Enum_With_Rep
…
end interface PSL::Core::Enum_With_Rep
...
type Day_Of_Week is Enum_With_Rep <Modular<2**7>,
        [#Monday => 1<<0, #Tuesday => 1<<1, #Wednesday => 1<<2,
         #Thursday => 1<<3, #Friday => 1<<4,
         #Saturday => 1<<5, #Sunday => 1<<6]>
```

Univ_Enumeration -- Invented or Discovered?

- Once we had established the general approach for allowing user types to make use of literals, the Univ_Enumeration solution was just out there waiting to be found
- Inspired in part by #t and #f of Scheme, ‘abc syntax for symbols in Lisp
- Seems to address extensibility concerns expressed by Wirth and Meyer
  - “from_univ” operator in extension can have a “weaker” precondition -- support more literals
- Preserved ParaSail’s “uniform syntax” goal for type as an instance of a module
- Benefited from deadline-free, blogging approach to design

Lessons learned

"... in contrast to a more systematic design-team-based or committee-based process, the blog-based process has allowed us to jump from one aspect of the language to another, allowing the author to go through more of a *discovery* process than an *invention* process. That is, rather than systematically tackling particular design problems and attempting to force a solution, problems were allowed to percolate in the background while effort focused on parts of the language where solutions were more immediately apparent. At some point, as part of experimenting or ruminating, a solution for one of the background problems suddenly emerged, as though it was always there but just was not yet visible. This has allowed the author to stay very close to the original design principles, rather than being forced into compromises to satisfy a schedule or other external requirements for steady progress...."

ParaSail Website and Documents

- ParaSail blog: https://parasail-programming-language.blogspot.com/
- ParaSail web site: http://parasail-lang.org
  - Programming Journal Issue 3.3, February 2019, *ParaSail: A Pointer-Free Pervasively-Parallel Language for Irregular Computations*
  - Embedded.com, *ParaSail: Less is More with Multicore*
  - *Designing ParaSail while Blogging*
  - Embedded.com: *Go, Rust, and ParaSail: Alternatives to C/C++ for Systems Programming in a Distributed Multicore World*
  - Talk at Microsoft Research, October 2012, *ParaSail: A Pointer-Free Path to Object-Oriented Parallel Programming*
  - CotsJournalOnline.com: *Parallel Programming Languages Enable Safer Systems*

History of ParaSail

- **2009 – Started design of ParaSail, on a blog**
  - My friends were too busy – drowned my sorrows in a ***blog***
- **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