Paws' Nucleus

Design-10

You can skip to the **tl;dr** if you like. =)

The plan.

Hi. I'm **ELLIOTTCABLE**. (Yes. In all-caps.)

This first-stab at a specification (collectively, for the **Paws' Nucleus** semantics, and the **cPaws** syntax) is a design for a *single element* out of a concerted whole, a much larger project encapsulating these two elements. I'm going to avoid waxing ineloquent on my higher-level goals here, but there's a few things you should know about the project as a whole to aid comprehension of the parts you *are* here to about to read about.

An Abstractive Target: First off, this language (the 'Nucleus') is designed as a sort of high-level virtual-machine, with semantics appropriate for an "intermediate representation" of sorts. The semantics you're about to read about is not something you write by hand (like Ruby); nor, however, is it a language you *translate* or *compile* into (like LLVM's IR, or in these modern ages, JavaScript.) Instead, it is a language you *abstract* into. Higher-level abstractions are not intended to be written in a new syntax and translated into some syntax only readable by this VM; instead, higher-level abstractions are intended to be folded onto the Nucleus, modifying both syntax and semantics incrementally, instead of changing them in one fell-swoop of compilation. When reading this spec, keep in mind that fact: that the target "user" of this language is the person writing abstractions **on top** of the Nucleus. This isn't intended to be an easy-to-use language for the end-user.

A Moving Target: Second, we're still at an early stage of the design. The language *as it exists now* is only being specified, *to further aid ongoing design-work*. Some that is here, will not make much sense without further design-elements within whose context it has been designed … and yet, those further elements have been purposefully omitted from this work. They're under too much flux, and require too much further effort before they can be tied in beautifully with what exists here. If you're going "why the fuck would …", it's probably because the *understanding* of that element of the design is predicate on one of the major design-goals which is not yet herein included. (To rephrase … if you're reading this, you know me, and you should just ask.)

So! Let's talk roadmap.
This, as described above, is a part of a larger whole. Let's examine that whole.

1. **Paws' Nucleus:** This. What you're looking at. An ascetic core-language, from which we've removed every feature, procedure, and datatype that isn't *absolutely* necessary.
2. **Paws' Core:** A light abstraction-layer on top of the Nucleus, wherein we implement many of those things we removed, *in terms of the things that were left*. (Think Rubinius.)
3. **Paws' Fullness:** The Nucleus and Core, taken together, are intended as a platform on which to design and implement a particular type of programming languages. (Highly abstracted, highly

expressive, and highly asynchronous languages.) Paws' Fullness is one such language, a very dynamic, powerful language that takes full advantage of everything Paws provides. (Nucleus is targeted at language-authors, but Fullness is targeted at *end-users*.)

You'll see no further notice of Core or Fullness in this document. Paws' Nucleus, the relevant portion, will simply be referred to as "**Paws.**" In addition to the separations between abstraction-layers, the Nucleus undergoes a further graduation, amongst 'capability levels.' (And, of course, versions.)

A given implementation of the Nucleus (that is, a VM, or host, for Paws programs), may adhere to one of three "compatibility levels." These allow host environments with reduced capabilities to still run Paws programs, but without particular facets of the design that prove problematic in some environments:

1. Paws' Nucleus, **basic** capability: A full, specification-compliant implementation of the Nucleus semantics, that does not propose to run code concurrently at any point (that is, a single-threaded implementation), and that does not implement any of the federation/distribution mechanisms (that is, a single-host implementation.)
2. Paws' Nucleus, **federated** capability: single-threaded as described above, but *also* a fully-functioning federating interpreter.
3. Paws' Nucleus, **complete** capability: both concurrent-execution capable (multi-threaded) *and* federating.

If you take everything in this specification, and translate it into code, you will have **#1** above; a basic-capability Paws interpreter.

tl,dr:

<3 You're reading an incomplete spec, for a *tiny fraction* of a much larger project. Don't get antsy. <3

Grammar

Rationale: Paws doesn't describe a single “syntax” for serializing Scripts (that is, Paws programs or procedures.) In fact, by design, some possible Scripts are *unserializable*! However, (mostly for convenience of testing, and communicating complex topics in Paws-related documents), we *do* specify a simple syntax that is capable of serializing a simple *subset* of possible Scripts. That syntax is called “canonical Paws,” or “**cPaws**.”

```
<expression>          ::= [ <word> { sp* <delimited-word> | sp+ <identifier> } ]
<word>                ::= <identifier> | <delimited-word>
<delimited-word>::= "(" sp* <expression> sp* ")" | <literal>
<identifier>          ::= identifier-character+

<literal>             ::= <symbol-literal> | <execution-literal>
<execution-literal>   ::= "{" sp* <expression> sp* "}"

<symbol-literal>::= '"' <symbol-content> '"'
<symbol-literal>::= '“' <symbol-content> '”'
<symbol-content>::= symbol-character*
```

- In the above, `identifier-character` represents any Unicode codepoint *outside* of Unicode's Separator and Other categories, excluding those used as terminals elsewhere in the grammar: `'(){}""“”'`
- Conversely, `symbol-character` represents *any* codepoint, *except* the codepoint necessary to close the `symbol-literal` that the production is found within.

Note that,

- the only place whitespace is *required* is to juxtapose undelimited literals; that is, `foo bar` is identical to `"foo""bar"`.)
- there is no definition for ‘lines’ or ‘statements.’ Paws doesn't provide any such semantic, so newlines are completely ignored.
- there is intentionally no way to serialize many Symbols into cPaws, such as the Symbol containing the closing-double-curly-quote *and* the double-straight-quote.

Examples:

```
foo bar baz            <L'foo' L'bar' L'baz'>
foo “bar baz”          <L'foo' L'bar baz'>
foo (bar baz) widget   <L'foo' <L'bar' L'baz'> L'widget'>
something { abc }      <L'foo' Ex{ L'abc' }>
something() { abc }    <L'foo' <> Ex{ L'abc' }>
```

Abstract types

```
interface Node
     readonly attribute {Expression|Object} word

interface Expression
     readonly attribute list<Node> words
```

Expressions in a Script are ordered sets of Objects (or other sub-Expressions).

```
interface Script
     readonly attribute Expression root
```

Paws' version of an 'AST.' A Script is a tree of Objects that describes a procedure for a Paws evaluator to follow. To be momentarily pendantic, unlike the traditional AST, it's neither abstract (since it references actual objects in the run-time of the program), nor syntax-related (as it's directly encoding the *semantic* relationships of the objects; specifically, combination.)

> This merits further commentary: as described above, Paws doesn't dictate a syntax. That's left up to the implementation. Instead, Paws dictates a semantic structure for instructions, in the form of combinations of *actual objects*. Meaning, where a normal programming language might have 'source code' with the word "dog" encoded in ASCII, Paws could instead have *the actual dog-object* embedded directly in its instruction-set.
>
> The common cPaws serialization format described above is converted into Scripts full of Label objects, approximating the traditional form of an object-less syntax-tree. In such a program (which is far from the only kind of Paws program), words, encoded as Labels, are used to *represent* concepts, as in a traditional programming language's source code.
>
> *(Not described in this version of the specification, as it is not finalized; but there will be a common binary format for exchanging constructed Scripts. In fact, this will be a subset of the format for exchanging entire, frozen, Paws memory-spaces / worlds; as any Execution encapsulates the Script being executed. =)*

```
interface NoObject : Object
```

This is a placeholder for situations where the concept of 'no result' needs to be conveyed. (Namely, in step **4d** of the `advance()` algorithm below; for the first Combination coming out of an Expression.)

```
interface Combination : tuple<Object>
     attribute Object subject
     attribute Object message
```

Combinations are described in more detail below; but this type presents the information necessary to *perform* a combination: that is, the left-hand side (the `subject`), or the thing being 'combined against'; and the right-hand side (the `message`), the thing being 'combined with.'

```
interface Mask
     readonly attribute Object root
     bool conflictsWith(Mask other)
     bool encloses(Mask other)
```

When perceiving the objects in a Paws program as a ‘graph’ (in the computer-science sense), a Mask represents a particular ‘sub-graph’ of that entire datagraph. By being given a starting-point (a Mask's `root`), and crawling the graph directionally exclusively along the edges marked by Relations' ‘ownership,’ we describe the boundaries of a particular “data structure” as it was intended by the programmer.

A Mask represents a subgraph of objects formed by taking one object and traversing the graph from that object, following only edges marked as `isChild`. *(If you're familiar with garbage collection, this is rather like a garbage collector working outward from a root by following strong references.)*

» `bool` **`conflictsWith`**`(Mask` **`other`**`)`

Given an `other` Mask, this function will return a boolean indicating whether **any** Object owned by this mask, is also owned by the `other` mask.

» `bool` **`encloses`**`(Mask` **`other`**`)`

Given an `other` Mask, this function will return a boolean indicating whether **every** Object owned by this mask, is also owned by the `other` mask.

Concrete types

```
interface Relation
     readonly attribute Object to
     attribute bool isChild
     Relation clone()
```

- At no point will Relations be modified; they're intended to be static members of an object. They'd as easily be implemented as a part of the object's data-structure.
- No one Relation instance will ever belong to *more* than one Object; when there is the same relationship-style link between *another* parent and this object, then the Relation can be copied, but not referenced a second time.

`» Relation clone()`

Creates a clone of the Relation in question, with the same `to` and ownership information. As a particular Relation cannot belong to more than one Object, they must always to be cloned before being added to another Object.

```
interface Object
     attribute list<Relation> members
     attribute Object receiver
     static attribute Execution default_receiver
     Object clone()
```

Paws' Objects are ordered sets of references to other Objects.

Rationale: While many languages have a generic Object-ish type described as either ordered lists (the LISPs), or as key-value dictionaries (JavaScript, Io), Paws has taken a somewhat unconventional approach. Our generic Object type is *implemented* as a simple ordered-list (à la LISP), we ***treat*** our Objects as if they're dictionaries. Specifically, the default receiver for Object makes some simple, tentative assumptions about the structure of the Object structures it traverses. (See the rationale under that heading for more information.)

Despite these assumptions, *all* other aspects of the design are intentionally structurally-agnostic. To boot, receivers are exchangable, and that structure-aware receiver can be overridden: this means that code written in Paws can replace the entire 'object system' (such as it is) with one of their own, as long as it can be expressed by unlimited nested ordered-lists (and LISPers have made a good case to support that `:P`).

`» Object clone()`

Cloning Objects is a fairly simple operation: a new Object is created, and then the `members`-Relations of the original object are cloned, retaining order, to the new Object's `members`.

`» Execution default_receiver`

The default combination-receiver for all Objects with no `receiver` explicitly set (except when overriden by default, such as for Executions, which have their own default-receiver.) Basically, this Execution encapsulates the algorithm that will be followed when a Combination is evaluated by the reactor, until an Object has its `receiver` defined.

The default functionality for Objects receiving a message, is to 'look up' that a value for that message in the contents of the receiving Object. That algorithm for the Object receiver, given a `params` Object containing receiver parameters:

1. Break the `params` Object down, taking the second `member` as the `caller`, the third `member` as the `subject`, and the fourth `member` as the `message`.
2. Iterate *in reverse order* through the `members` of `subject`, looking for the latter-most `member` in the set that satisfies both the following criterion:
    a. There is an extant Object, `value`, in the second slot of `member`'s own `members`
    b. That `value` has a identical object-identity to `message`
3. If no such matching `member` is found, then the algorithm terminates with no further actions.
4. If such a matching `member` is found, then we take as the `result` the value in the third slot of that `member`,
5. … and queue the `caller` with the `result`.

Rationale: This algorithm assumes a flexible data-structure involving 'pair' objects, with a second member acting as 'key,' and a third member acting as 'value.' A parent Object containing a series of these 'pairs' thus acts as a fairly efficient key-value dictionary.

Notes:

- If the lookup does not result in a value, then the `caller` is never resumed. This is intentional, and amounts to a (brutal) 'Paws boolean.'
- `message` may be *any* kind of object; however, it is often a Symbol. Hence why the Symbol is designed for optimal runtime comparisons; acting as the comparative 'key' in this algorithm is its primary purpose.
- This algorithm obeys the 'noughty' rule, particularly with respect to the pseudo-pairs it encourages. That's why it expects the key and value to be stored in the second and third slots, respectively.

`interface` **Symbol** : **Object**

```
    readonly attribute string name
```

Symbols are a static type, read at compile-time and compared by identity (not content). Think Ruby's or Lisp's.

- A symbol with a given name will only exist *once* within the system. All symbols with equivalent name will have the same object-identity.

Rationale: There will be no user-facing method for *creating* Symbols electively, by design. The majority of Symbols in a runtime are expected to come into the program via the Script. There are, however, ways for new Symbols to get into a particular unit of evaluation (exploding an existing symbol which contains characters that don't currently have their own extant Symbols; also, theoretically, federation with other units.) Thus, a system is expected to compile Symbols into some sort of easily-comparable-at-runtime format, but it should still allow for generating new identity-comparable objects at runtime.

```
interface Execution : Object
    attribute Script root
    attribute bool pristine
    attribute Node pc
    attribute list<Object> stack
    static attribute Execution default_receiver
    Combination advance(Object response)
    void store(Object value)
    Object retrieve()
    Object pop()
```

- Although they are similar to the more traditional “continuations” from programming-language theory, our executions are *not* static. One does not simply take a execution, and then have a handle to resume execution at the point it was taken indefinitely. When a particular execution-object is used to resume execution at that point, the object itself “moves forward” **with** the procedure's execution. (However, if one wants a static reference to a particular position in the program, they can take a execution and then clone it, taking care to ensure their clone is never called.)
- Although `pc` is not *`readonly`*, the implementation is expected to ensure it can only ever be “moved forward.” That is, although our executions act as mutable program-counters, they only ever move *with execution*.
- When created, Executions' `stack` and `pc` will be non-existent; and `pristine` will be `true`.

» `Combination advance(Object response)`

An Execution's `pc` is intended as a reference to “where it is” in the relevant Script. This method moves the `pc` forward one step, given a `response` for the *last* Combination in the Script.

1. If the `pc` is non-existent, the `stack` is empty, and the Execution is no longer `pristine`, then we may presume that this Execution has been completed, and short-circuit this algorithm with no product.
2. As the `pc` denotes the *last* node combined, the `current` node we're generating a Combination for will be defined as the next Node *after* the `pc` within its immediately-enclosing Expression.
3. If `current` is undefined (i.e. there was no Node after the `pc`, because `pc` was the last node in the enclosing Expression), then we take `current` to be that *enclosing* Expression (`pc`'s parent), and produce a Combination of the product of this current expression, against any earlier-stored products:
    a. We update `pc` to that `current` enclosing Expression, indicating that the Expression has been completed
    b. Now we `pop()` the product of the last combo prior to entering the now-completed Expression off the `stack`, and store that as the `subject` of a new Combination …
    c. … and add the `response` of the previous combination (provided to this algorithm) as the `message`.
    d. then terminate this algorithm, with that Combination as the product.
4. Else if, and as long as, the new `current` refers to an Expression, we'll begin *repeating* the following:
    a. If `current` is an *empty* Expression (i.e. its `words` is an empty set), then this node is a special case, and implies a reference to the *enclosing Execution*. Short-circuit this operation, and jump to **5**; but treating the `current` Node as one containing the current Execution *as an Object*.
    b. We increase the length of the `stack` (leaving the new slot at the top of the `stack` ‘empty.’)
    c. If this is the *first* iteration of **4**, then we `store()` away the provided `response` in the new slot at the top of the `stack`.
    d. We remove the provided `response`, replacing it with `NoObject` for the purposes of step **5**.
    e. Finally, we dive into the sub-expression by updating `current` to the first `word` of the old `current`
    f. … and if the new `current` is also an expression, continue again from **4**.

5. `current` now refers to a node containing an Object. We update `pc` to match `current`, and construct a Combination with the provided `response` as the `subject`, and the Object-contents of `current` as the `message`. That Combination will be the final product of this algorithm.

Rationale: The above is a complex way to say the following:

- An right-hand-side Object is combined against the result of the *previous* combination.
- When a sub-expression is encountered, the result of the previous combation is put on hold, pending the completion of that sub-expression.
- When a sub-expression is completed, the result of the sub-expression is combined against the stored result adjacent to it, as if the sub-expression had been replaced by its last result.
- The *first* Object in an expression (or sub-expression) is implicitly combined with nothingness (conveyed as `NoObject`); this is handled in the reactor as an *implicit* combination with the running Execution's `locals`, although that is not of interest here.
- An entirely empty Expression is special-cased as a reference to the running Execution. (Think `this`).

(For the next revision of this document, I'm planning to refactor the monolithic and rather disgustingly-tangled `advance()` into multiple, more dis-entangled functions; but until then, these are the most I could tear out into their own concerns.)

» `void` **`store`**`(Object` **`value`**`), Object` **`retrieve`**`(), Object` **`pop`**`()`

`store()`, `retrieve()`, and `pop()` manage the `stack` for `advance()`.

- `store()` *replaces* the top-most element of the `stack` with the provided `value`.
- `retrieve()` returns the top-most element of the `stack`. If the slot is empty, that will be `NoObject`.
- `pop()` removes the top-most slot from the stack, `retrieve()`ing the element that previously occupied it. (Thus, making the second-from-last element the new last element.)

» `Execution` **`default_receiver`**

Executions without a defined `receiver` don't use the Object's default-receiver; that is overriden with their own default. Instead of preforming a 'lookup' against the data-values stored on this object, we treat a combination against an Execution (as usual, barring an overridden `receiver`) as a procedure-call:

1. Break the `params` Object down, taking the second `member` as the `caller`, the third `member` as the `subject`, and the fourth `member` as the `message`.
2. Then simply queue the `subject` execution, with the `message` as the response.

Semantics

Definitions & algorithms

Ownership

Paws requires the programmer to describe the boundaries of their data-structures, giving the runtime the hints it needs to decide which sections of code it can safely paralellize. These hints are called ‘ownership,’ and are very simple: the programmer explicitly states, within a data-structure, whether it *owns* another data-structure it links to.

This is implemented as an annotation on the Relations stored in an object, specifically, `isChild`. If a programmer is constructing a compound ‘data structure’ out of multiple, nested objects, then the programmer is expected to flag each nested object as owned-by its parent object with `isChild`. Conversely, whenever the data-structure *references* another, separate data-structure (the canonical example being a ‘dog object’ and its ‘person object’), the `isChild` is left un-set.

Rationale: Data-structures constructed in Paws programs are intended to comprise hierarchies of sub-objects. Paws' parallelism and asynchronicity system depends upon knowledge of *which parts* (or ‘subgraphs’) of the data involved in a Paws program are self-contained data-structures; that “map” of the data-graph, describing which areas thereof are self-contained items of interest, is used to intelligently prevent concurrent access to data being elsewhere manipulated.

This information is encoded into that data-graph via the mechanism of *‘ownership.’* A structure (that is, an object, or nested group thereof) in a Paws program ‘owns’ an object, if the parent-structure has a Relationship pointing `to` the child-Object, and that Relationship is annotated as `isChild`; *or* if the parent structure owns a structure that, in turn, owns the object in question. (Thus, ownership cascades through the grah of data in a Paws program until it reaches a Relationship that is not `isChild`.)

Example: Imagine a data-structure resembling the following, as represented in a JSON-esque format:

```
person: {
     name: { 'Elliott', 'Cable' },
     age: 24,
     city: { 'Chicago', 'Illinois' } }
```

Given this example, we'll posit that the top-level object `person` ‘owns’ the `name` object containing `'Elliott'` and `'Cable'`. That is, we the programmer, are expressing that should the `person` object find itself being modified, it's likely that the `name` object would be modified as well. In Paws' estimation, that means that the `name` structure is a sub-object of the `person` structure, and that information should be encoded in the link between the two objects as ownership; that is, we would say that this `person` object owns its first member. This would be implemented as a Relation with a truthful `isChild`.

Conversely, the `city` object is data unto itself: a process modifying the `person` is not likely to ever need to modify the `city` (no change to Elliott is going to change the fact that Chicago is in Illinois). Thus, we can expect that our `person` object would not be encoded as ‘owning’ its third member.

The ‘noughty’ rule

The design of higher-level abstractions on top of Paws' Nucleus necessitates a useful place to store metadata beyond the absolutely paltry selection that this design provides natively. To allow for the encoding of meta-data about objects, we *intentionally* guarantee that no part of this nuclear design will ‘touch’ the *first* (that is, index 0 …

thus, “nought-y”) slot in an Object. Algorithms neither ignore it (thus being compatible with zero-indexed abstractions, if that's to the implementer's taste) *nor* expect it to be usable/available (thus being compatible, also, with clearly-superior one-indexed designs. ;)

A word on indices …

With the ambiguous, agnostic approach to indexing we take in Paws' nuclear design, the meaning of terms such as ‘first’ and ‘initial’ can sometimes get confused. To avert this, throughout this specification, we're going to use the following terms to mean *exactly* what's dictated here:

- *first:* The item at index **0** in given Object; that is, `members[0]`
- *second:* The item at index **1** in given Object; that is, `members[1]`
- *third:* The item at index **2** in given Object; that is, `members[2]`

… and so on. (If intending to refer numerically to an indexed object, instead of using ordinals, we'll use an explicit cardinal: that is, we may refer to ‘member two’ or ‘member zero’, instead of ‘the third member’ or ‘the first member’, respectively.)

Combinations

A Paws program is composed of only one basic operation: the 'combination.' It is composed of a message (one object) being sent to a subject (another object.) The *semantics* of the combination operation are overridable, by replacing the receiver for an Object; that's how most of the semantics in a Paws program are implemented.

As expressed in the abstract object-model above, in a Script (a Paws program,) there are Expressions formed of individual words. That series of words *represents* a series of these combinations. (This is explained more in the rational section under Script.) Every program tick in a Paws reactor, one of these combinations is picked from that Script (see Execution's `advance()`), and then evaluated to produce a resultant single Object.

When a message is received by an object, both the message and the object (as well as the Execution during which the combination was being preformed) are passed to the Execution registered as the object's `receiver` (composed as a receiver-parameters object.) The default receiver-handler differs between different kinds of objects (see Object's, and Execution's, above.)

Notably, for most objects, combinations default to what is effectively a key-value look-up of the message, on the subject. (Again, see Object's default receiver.)

Combinatorial algorithm

Composing a value of a combination is the core task of a Paws machine. Given a `subject` and a `message`,

1. Acquire a receiver for the `subject`.
2. Construct a `params` object with the current Execution, the `subject`, and the `message`.
3. Call the receiver with that `params` Object.

Notes: The current execution is implicitly stopped after each combination; that is, it has been processed, and then the call-pattern invoked in step **3** of this algorithm leaves it stopped. The receiver is usually expected to re-queue the current Execution (the 'caller') passed to it, in most cases. That's intentionally left out of this algorithm. However, for objects with a built-in `receiver` that is *known* to immediately re-queue the caller with a result, there's an expectation that performant implementations will process multiple combinations in series, without ever removing the caller from the reactor. The specifics of this are left up to the implementation.

Similarly, the *result* of a combination is not of interest to the Paws runtime itself, strangely enough. It's up to the receiver to do something with that (usually, presumably, providing it to the caller in some fashion.)

Finding the receiver for a given Object

A 'receiver' is an Execution associated with a given object, one responsible for handling combinations when that object is the `subject` of the combination.

1. If the `subject` has no `receiver` property set, then an Execution implementing the "default receiver" algorithm for that type of object is the result of this algorithm. (Each type described above includes a description of that type's default receiver's algorithm.)
2. If the `subject` has a `receiver` property, and the value of that property is queueable (i.e. an Execution), then that Execution is the is the result of this algorithm.
3. If the `subject`'s `receiver` is not queueable (that is, not an Execution), then recursively apply this algorithm starting at **1**, with that `receiver` as the `subject` *for the purposes of this algorithm*. (*Not* for the consumer of this algorithm, who will have their own reference to the original `subject`.)

Rationale: The recursive nature of this process allows object-system designers to wrap their receiver(s) in metadata, or otherwise abstract them away.

The parameters-object passed to receivers

You get a nasty recursive mess if you try to coproduce parameters into the receiver on every tick, which is otherwise the most common approach to parameterization in Paws; instead, we take receivers as a special-case, and compose all of their arguments into a single object, then pass them *that* instead.

The object passed by the reactor to an Execution registered as a receiver comprises the following (none of them owned):

1. An *empty* first slot.
2. The `caller` (that is, the currently-running Execution at the time the combination was queued), in the second slot.
3. The `subject` in the third,
4. and the `message` in the fourth.

The Receiver is intended to do the deconstruction of this argument object itself.

Reactor operation

The Reactor

A Paws machine's ‘reactors’ are processors that choose an execution from a queue of waiting executions, ascertain the next combination dictated by the state of that execution, and then evaluate that combination. These simple ‘ticks’, described in detail below as the ‘reaction algorithm,’ are the heartbeat of a Paws program. Every single action or algorithm in the language boils down to a long series of combinations processed in that way.

(This version of the spec isn't fully *verified as concurrency-capable; but nonetheless, multiple Reactors are even now designed to be run in parallel.)*

The reaction algorithm

The basic ‘tick’ in a Paws reactor, is to choose a ‘safe’ Execution, and evaluate the ‘next’ combination from it. This happens as follows:

1. The next execution that can be safely advanced is chosen from the queue is determined by the reactor predicate, and it and its taken as `current`, along with its associated `response` and (if any) `mask`.
    a. If no execution is found (i.e. the reaction queue is empty, or everything in it failed the predicate), then this tick is aborted
2. If there's a `mask` request associated with the now-`current` execution,
    a. … the `current` execution is granted responsibility for that `mask`.
3. Get a `combo` by calling `advance()` on the `current` execution, with the `response` as argument.
4. Preform the combinatorial algorithm on the `subject` and `message` from that `combo`.

Note: Once an execution has been provided to this algorithm by the predicate, it's already been *cleared* for any mask-request it might have. That means that step **2** can simply grant the responsibility requested without any further verification.

The reactor predicate

Multiple reactors are intended to operate concurrently, processing combinations from mutually-unrelated parts of a program. To do so safely, only executions that are ‘safe’ to continue evaluating are chosen from the reaction queue. The following algorithm will select a safe execution from the queue:

1. Iterate through the entries in the queue, extracting the `queuee` execution, the `result` object to be fed to the `queuee`, and the `mask` being requested (if any.) For each `queuee` in order,
    a. If the `queuee` is already being processed by another reactor, skip it entirely and proceed to the next entry in the reaction queue.
    b. If the `queuee` *doesn't* have an associated `mask`, then its been allowed to begin running whatever section of code it is in now on a previous pass through this algorithm, and thus it's guaranteed to be safe. That `queuee`'s entry will immediately be the result of this algorithm.
    c. If there *is* a `mask`-request associated with this `queuee`, then if *either* of the following is true, that `queuee` is the result of this algorithm:
        i. There is an existing mask for which `queuee` is already responsible, which the requested `mask` `encloses()`.
        ii. The `mask` requested is available to `queuee`.

Notes:

- Step **1** is necessary, because a single Execution object can appear in the reaction queue multiple times. If one reactor consumes one instance, we want to make sure we don't end up with undefined behaviour

where a second reactor is trying to obtain another combination to process for the same Execution, at the same time.

The reaction queue

At any given time in a running Paws program, there's a set of executions waiting to be (further) evaluated by a reactor; this set is called the ‘reaction queue’ (or just ‘the queue.’) Each entry in the queue is associated with the other information a reactor will require to process it: the ‘response’ of, and, if relevant, a mask describing a section of the program's data-graph (memory-space) that the execution in question is waiting on responsibility over.

The design of the queue's consumers guarantees a few things:

- #INCOMPLETE

Queueing executions

To be queued, an execution needs a value to fill the ‘hole’ left by its previous combination. This is called the ‘response,’ as it is usually the result of an external computation the execution was waiting for.

An execution in the queue can also have ‘requested mask’ associated with it; this describes a set of responsibility that the execution is waiting for.

The ‘call’ pattern

#INCOMPLETE

Responsibility

Responsibility is the fraternal-twin to ownership: it's the system by which a reactor tracks *which blocks of code* can safely access which ownership-delinated data-structures.

A Paws machine must keep track of the responsibility granted to any code at any given time, in the form of masks paired to the executions that requested them. While absolution of responsibility may be enacted from anywhere, and is pretty straight-forward (either removing a record tying a particular mask to a particular execution, or removing *all* records for a particular execution); it's important that only a reactor, having verified the reactor-predicate for a given responsibility-request (in the form of a mask attached to an execution queued to be evaluated), ever *add* things to the responsibility tables.

When an execution needs to acquire responsibility (or, equally likely, some responsibility is requested *for* it), it is attained by being queued with the a mask describing the sub-graph (that is, the data structure(s)) as the requested-mask associated therewith. The mechanics of this being checked are handled by the reactor predicate.

Rationale: This isn't just a tool for managing memory-safety in a concurrent environment; in fact, much of *that* is already required of the implementation, and not something the programmer should be worrying about (relevantly, some implementations may not actually *be* concurrent, and won't have to worry about that at all.) Instead, responsibility is also a tool for the basic ordering of data-operations in an asynchronous environment. A block of code that acquires responsibility for a data-structure is stating “I am going to preform several *sequential operations* on this data, which must not be interleaved with other modifications;” and it's our job to make sure nothing else that may also want to access that data violates that pact once we grant it.

Responsibility primarily operates on masks: as masks effectively delinate data-structures in the memory-space, or ‘data-graph,’ of the program, they are the ideal tool for discussing who is *responsible* for a particular data-structure. (And, since the meaning of a mask is dependant upon the ownership of the object(s) it references, this is how responsibility depends upon ownership.)

Recording responsibility

When a reactor is ready to grant responsibility for a `mask` to a given `queue`, having satisfied the requisite predicate, the following algorithm is observed:

1. If there are any `existing` masks registered in the responsibility table for `queue`, we iterate over them,
    a. checking if the requested `mask` `encloses()` that `existing` one,
    b. … and if so, absolving that entry.
2. Finally, an entry is added, pairing that `mask` with the `queue`.

Availability of responsibility

Before a `queuee` execution can be given a `mask` that it is requesting responsibility for, it must be determined if responsibility for that sub-graph isn't already taken. The following algorithm determines whether or not this is the case:

1. If there are any `existing` masks registered in the responsibility table for `queuee`, we iterate over them,
    a. checking if the `existing` mask `encloses()` the requested `mask`,
    b. and if so, terminating this algorithm, indicating availability.
2. Else, iterating over any `other` masks (that is, masks registered for executions other than `queuee`),
    a. checking if the `other` mask `conflictsWith()` the requested `mask`,
    b. and, if so, terminating this algorithm, indicating unavailability.
3. Finally, if no conflict is thus found, we indicate availability.

Programs in Paws

The Paws Machine

A Paws machine (that is, an interpreter, or some other sort of runtime, implemented according to the above spec, and thus capable of processing combinations) is thus expected to encompass the following portions:

- A data-graph, comprising a set of Objects containing directed references to one another.
- At least one Script, describing combinations to be preformed.
- One, or more, reactors. To process those combinations (as extracted from Scripts via Executions.)
- A reaction queue, maintaining *ordering* information for operations yet-to-be-preformed.
- A responsibility table, maintaining *locking* information on operations currently in progress.

A Paws Unit

All of the information necessary to describe the state of a Paws program *when between ticks*, is sometimes called a 'unit.' (Alternatively, everything listed above as the elements of a Paws machine, minus the reactors themselves.) Although it's not provided for in this document, there's no reason why a pool of reactors and a chunk of memory couldn't be shared by a set of more than one Unit.

Frozen Units

More interestingly, there's nothing in specific that ties a Unit, when it is between ticks, to the set of reactors it's running on; or, to be specific, one could spin down all of the reactors, allowing them to complete their combinations (but denying them new combinations to evaluate), and then record the state of the rest of the machine: the data-graph, the reaction queue (and thus the Script(s) being executed, via the Execution objects in the queue), and the responsibility table.

The product of this is known as a 'frozen' unit. Eventually, a binary exchange format for these constructs will be specified; but until then, it should be perfectly easy for a particular implementation to export such a frozen unit to a proprietary format, and re-load, programs in this manner.

Rationale: Although one can exchange proto-code (that is, some form of unaccompanied Script, or something that constructs a Script, and which must construct its data-graph from scratch) in the traditional manner, i.e. in the cPaws format laid out at the beginning of this document, the author thereof believes that there's many interesting things to be done with the distribution of entire frozen programs, programs that have already de-constructed themselves into data-graphs, *with* any further associated Scripts. More on that, in a future version of this document. ;)

('Units' are also *the key to federation; that is, Paws machines connecting to act as a distributed system; but that's left for a future version of this specification to cover.)*

Constructing Scripts from cPaws

Since any programming-language interpreter implementation is probably going to start with a parser, I suppose I should wrap up this document with these notes, to allow you, my dear reader, to proceed from this point into writing actual code for your *very own* Paws machine.

cPaws' syntax closely mirrors the structure of a Script, by intention. In fact, approximately all it is is a format for encoding nested Scripts, with inner Scripts being initialized into Executions inserted into the parent Script.

When parsing a cPaws file, to produce a Script, you need simply:

1. Create an Expression for the contents of each *<expression>* encountered
2. Create Symbol-containing Nodes for all *<identifiers>* encountered within the expressions,
3. … as well as for the *contents* of all *<symbol-literals>*.
4. When encountering an *<execution-literal>*,
    a. recurse this entire process;
    b. then create a pristine Execution of the resulting Script,
    c. and insert that Execution into the Expression generated for the parent of the *<execution-literal>*.

You'll end up with a Script full of (sub-)Expressions, themselves full of Symbols and perhaps a few Executions. That's a Paws program!

<3 Now go have fun, kids! ***Do*** try this at home! <3