2025 Progress Update: Towards A Universal Language

An initial look at the text-based
Ray programming language
and subsequent design notes for its IDE: The Ether.

OrbitMines Research

Fadi Shawki
fadi.shawki@orbitmines.com

31 December 2025

Discussion Channel

Introduction

After several years of abstract thought [1] [2] [3], actualization is the next step in the designing of a kind of universal programming language. The central question being: how do we evolve programming languages and their respective compilers and ecosystems forward?

This is a bit of a technical update on the state of the ideas I'm working on to combat this question.

Arc: The Ray Programming Language

A new language

I'll start this excursion from the perspective of a new text-based programming language. Though this project intends to step away from the limitations of the text file, all programming infrastructure relies on it. A move away from it, will require additional infrastructure. Even if this is achieved, being able to express as much as possible in a traditional text-based format will be beneficial. Though there will be design features which are simply not translatable to a purely text-based programming language.

Every variable...

Every variable... is Many

Even though most compilers use some form of abstract interpretation, a language which natively supports superposed values is certainly unusual. You only really see it used in type systems. There do exist implementations of something like an 'Ambiguous Operator', but it is not as expressive as it is for one of the cornerstones of the Ray language:

Take for instance the following boolean values:

```
false | true // (= boolean)
false & true
```

These are not boolean OR and AND operations. Instead they superpose possible values for that particular variable. In the | (OR) case, it's: this value is either false or true, but I don't know which one: This is just like the boolean type.

Since they are castable to boolean, you can call functions accepting a boolean with them:

```
s (x: boolean) => x ? "Y" : "N"
s(false & true) // "Y" & "N"
s(boolean) // "Y" | "N"
```

This too, works for defining and calling superposed methods. Whether it is to combine the results of many methods, or whether it is to function as multimethods. For example, if we superpose the boolean operators AND and OR:

```
true (|| | &&) false // (true || false) | (true && false)
```

Or for the multimethods case, only methods matching the parameter types get executed: (Note that you can of course, also give multiple names to the same function, as if defining aliases)

```
a | a1 (: boolean) => "X"
a | a2 (: Number) => "Y"

// a is (a1 & a2)
a(boolean) // "X"
a(Number) // "Y"
```

In fact, this is even as powerful as to extent to possible implementations of a function. Take for instance the way boolean operators are defined in Ray. They are all recursively defined in terms of each other. The NOT gate has definitions like:

```
!{.}
 | this !&& this // nand
 | this !|| this // nor
 | this x|| true // xor
 | this x!| false // xnor
```

All other operations would have something similar. What this allows you to do, is say things like: I don't know which one is supported by the system it eventually ends up running in, but I know how to get from one to the other.

Each with a different performance profile. One perhaps serving as specification of the algorithm, the other one focussing on performance. The compiler would in turn decide which one to use.

It's also good to know that this sort of thing works for any part of some iterable structure, albeit an array or graph. Take a string for instance:

```
"A", ("B" | "C") // "AB" | "AC"
```

All this arbitrary structure is accessible through the # operator on a variable:

```
x = false | true
x# // Access iterable structure
x#.count // == 2
```

This takes care of an important requirement for a universal language, namely: "I want to be able to say: Whenever you have one of something, what if you had more of that thing.".

Every variable... is a Ray

Instead of branding a language's abstractions as inaccessible. The approach of Ray is slightly different: The meaning of every abstraction must be accessible. Whether that's control-flow of a function, or the structural definition of a number. In a quick way of phrasing it, you achieve this by saying that "Everything is a kind of Structure/Graph" and that structure must be accessible. Or the term I'm using for it, since the approach we're using here will be more general than Graphs, is: "Everything is a Ray".

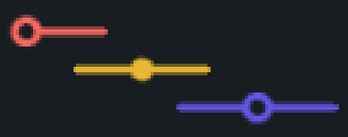
(Then phrasing inaccessible abstractions just becomes: There's structure there we're ignorant of. We can simulate this by ignoring structure.)

A good place to start is to understand how this graph-like structure I'm calling a Ray is defined. Which is simple enough to understand, especially if you're already familiar with graphs or linked lists.

Essentially it's nothing more than being at a point, ..., vertex and having information on what's in front of you, and behind you. If we visualize that point like this:

Then in front and behind, we define an initial and terminal boundary:

Each boundary then in turn optionally defines other boundaries, together they make an
edge
. (And if there's no additional boundaries defined, it's a
dangling edge
; or an actual boundary of the structure.)



Then of course, at that boundary, another vertex is defined.

You can keep repeating that and here we have the familiar structure of an Array. Which is simply defined as a line (of points).

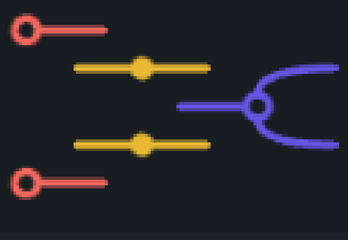
Where of course it gets just a little more complicated, is when we take into account what I said earlier: "Every variable... is Many". You'll see in this case, that instead of an Array, that the ideas of graphs and hypergraphs fall naturally out of that definition. There are 4 places where we are defining variables here, namely:

(1 & 2) Each
point
 has many
initial
 and
terminal
 boundaries. Or in other words, they define many
edges
. This upgrades our Array to the definition of a
Graph
.

Then the next two, are ways of upgrading our Graph into a Hypergraph. By turning the edges into hyperedges. Note that 'hyper-', might as well stand for 'Many'.

(3) Each boundary defines many other boundaries. (Which is the typical definition of a hyperedge)

And (4) each boundary is connected to many vertices.



Because the only differences between Arrays, Trees, Graphs & Hypergraphs is what kind of edges are defined on the boundaries, we get the following property in our language (without explicitly creating an object hierarchy):

```
Array
 ==.instance_of Tree
 ==.instance_of Graph
 ==.instance_of Hypergraph
```

And then the last few pieces to make it all fit: How do we know that there are Many defined, instead of one? It's recursively a Ray: Whenever you have Many of some variable, what you actually have is an iterable structure called a Ray, which defines on each of its points what's defined there.

Take the edge we're defining here for instance:

What this actually is, is some other structure, with three entries in it; one for each of the boundaries. Together this structure makes the edge. And essentially we're saying: "These boundaries are equivalent along this ray".

As you can see there: Every vertex, boundary and edge. Has a value on it. What type is that value? You guessed it, it's another Ray. Just like how we just defined the edge: At each of the vertices there is a boundary. And since every variable is Many. The value defined on each point is actually many Rays.

Which brings me to the last piece of what a ray is: An important part of structures like grammar or programs, is the ability to group together many parts into a single entity. A ray, can also be expanded and collapsed to reveal or hide structure. Take an initial boundary for instance, we could hide additional structure in it:

Collapsing that additional structure inside the boundary, would give us:

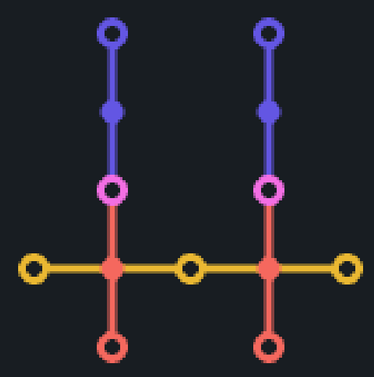
This hidden structure doesn't need to be adjacent, but often is. Another common use of it, is a parallel structure, in the sense that it is 'the same thing on another level of description'.

In the next sections you'll see how this all is used to encode structure like a binary number, or to encode programs.

Every variable... is a Type

As you now know, a normal variable allows you to construct types. That makes it possible, with regular syntax, to support things like dependent types.

If we take for instance a 2-bit binary number (00) which intuitively looks something like:



We might have a type requirement of one of the methods on the number, take the length of the number for instance, which in this case would be two. We'd check for that simply with:

```
Binary{length == 2}
```

You can also extend the type systems with arbitrary asserts, which add additional constraints, just like this dependent type would.

```
class Example < Binary
 dynamically assert length == 2
```

Which would be helpful when you have conditions which depend on multiple variables in intricate ways.

What is unusual about the type system, is that you can use define arbitrary patterns which must be matched when extended. (In the sense that, for example some valid regex syntax (type) matches certain string instances.)

An example of this sort of
type, pattern
 can be seen in how
IPv6
 is implemented. Where there are two complications to a valid address: (1) a sequence of zero segments can be compressed with '::' and (2) an
IPv4
 address might be embedded in them. Making a valid address something like
::ffff:0.0.0.0
 or
64:ff9b::
.

```
class IPv6 <
 (left: Segment[]).join(":")?,
 zero_compression: "::" (? if !defined_segments.empty),
 (right: Segment[]).join(":")?,
 (":", embedded_ipv4: IPv4)?

 defined_segments: Segment[] =>
 left, right, embedded_ipv4 as Binary

 static Segment = Hexadecimal{length ⩽ 4}
```

The additional constraints of what all the different segments of a valid address must be are then implemented with asserts. If you're interested in seeing that implementation,
look here
.

Every variable... is a Lazy Expression/Program

As a bit of an unusual feature for a programming language, the expressions, subexpressions and control-flow of a program are all (if wanted) exposed to the runtime. Instead of having a usual AST, the control-flow of the function is what is exposed to the compiler and runtime.

Having the control-flow as the primary language for functions/programs, or in other words: just a graph (or rather a ray), should make it much easier to abstractly compare the functionality of programs. Where when possible, any subexpression is expandable to its control-flow. With the only primitive being conditional edges: So in text-based terms essentially just goto's wrapped in an if-statement.

Which is another requirement if we wanted to create a universal language: We need to be able to have some common language in which we can compare and reason about all types of programming languages.

The ability to talk about the control-flow natively, allows us to construct types like this, to require functions which terminate:

```
Program.Terminating
```

Which is equivalent to the following code which checks that all branches are of a finite length:

```
Program{expanded.length#.every ≠ ∞}
```

Of course, we can't guarantee that this type-check will halt itself (Halting problem). But we at least have the language to express this sort of thing when we do need it.

Note that this way things like
coroutines
/
multithreading
,
return
/
if
 statements, ...,

for
/
while
 loops all become then just a matter of structures in the graph. The way they are implemented then, is just to get the access they have to the control-flow of the function, and to edit it.
To see how all this is implemented, see here.

The simplest example would be a goto, since that is just an edge to some next part of a program.

First, like labelling in some programming languages, you can get a pointer to somewhere in a function by labelling it. So a simple A + B, allows you to:

```
label1\ A (label2\ + B)

label2 // A program/function pointer
```

Or you can get the current function with &, and iterate through the control-flow to similarly get a particular pointer. Each successive part is also just a program: Just instantiated at a different point along the control-flow. Note that this is an unusual feature: Usually any reference to a function is just to it's starting point, but since we allow access the the internals of a function, we can do this.

```
&next
```

Combined with those two things the goto implementation is rather simple. We get the program which called the goto function, and just alter its control-flow:

```
goto (program: Program)
 &caller.push(program)
```

Having covered that, we can start turning towards another powerful feature of the language: Every variable is a lazy expression/program/function. The current value of any variable is accessible through the following operator:

```
variable**
```

This gives us the powerful ability to access the functions which might still need to run to fill this variable, it gives you access to possible intermediate results if the function hasn't yet been completed.

I say current value, because that brings me to the next unusual feature of the language:

Every variable... holds a History

Every variable holds a history. (Or at least, it's available to the compiler should you want to use it, keeping and storing every bit of variable change is of course way too expensive. A more intelligent approach is required.)

```
variable% // a Commit
```

When you think about it, there isn't much of a difference between the unresolved lazy program and a history we want keep. The implementation of history, is therefore just a program under the hood. Both are just graph of steps to perform to arrive at certain states and branches.

```
class Branch < Program
 static Version | Commit = State
```

What this allows us to do, is to natively support version control over arbitrary data structures. We can have localized subexpressions which have different histories,

```
var = A, B, C = "ABC"
B = "2" // var == "A2C"
C = "3" // var == "A23"

B% == "B", \"2"\ // A ray with "2" selected
```

or group them together in a 'global order'; this ordering/grouping of changes is referred to as a repository.

```
var% == B%, \C%\
```

The hope of having nested repositories over arbitrary graphs (/rays), and this native support for version control, is that we can start having better infrastructure than just Git, which really only works for text files.

One particular feature of the IDE, The Ether, will be to make use of this in the following way: By versioning functions (as a repository), and possibly sub-expressions of a function, we can start referencing function versions in our network stack; communicate what version I'm running on, what kinds of patches are available for any particular functions. As infrastructure it will allow us to reason about this more, which will become quite valuable.

This also allows us to intertwine the IDE's features of version control (so keystroke history) with the actual repository it's in. Since keystroke history, is just a branch off the current branch, which we haven't yet merged.

Where we act as a database beyond version control, type conflicts(/changes) between versions which are detected will also require resolving - which would be done with patches to the database if they're not backwards compatible. So in this way merge conflicts will be expanded with additional features.

Instead of always forcing all histories to always agree on the order of events, or even which events, we'd also want to support those that don't. An example would be a message history, where each local instance might have a different ordering of events, but we still push to all those different orderings. But I don't yet know how that would look like from the language's perspective.

As a last thing, it's also worth noting, that any spatial changes, so for example adding a character to a string. As a local change, is a history on an edge, instead of a vertex.

Every variable... has Access Permissions

One of the requirements of the language is also to be able to communicate as a database. Whether that is to function as a new version control system, a networked file system or to serve packages, game files, packets or anything else you can think of. Which brings into view the following feature: Every variable has access permissions defined.

These access permissions allow you to do the usual private/protected/public shenanigans of a programming language. In the Ray language we simply use 'internal' or not use it.

```
class Example
 internal variable
```

How that is implemented, is that the filter we used earlier for dependent types can also be used in this way. It's equivalent to this code:

```
class Example
 Node{==.instance_of Example} variable
```

Where the Ray language differs, is that the possible recipient of this filter might be a Character: Player (think user), or NPC (think server or agent).

We might say that we want everyone, by which I mean EVERYONE with an internet connection to me to be able to access the variable:

```
@public variable
```

Which would be used by a public API or endpoint. For instance, a publicly available profile, or a distributed chatroom.

There is a single Character associated with a runtime. We might want to specify that character, and only on this local machine is allowed to access it.

```
@local variable
```

Similarly there are modifiers for Characters which are running on the same machine:

```
@localhost variable
```

Or we might want to say, only this Character, but on any machine which is running it. This would for instance be any information only available to you, the central Ether server, and any private servers you might have which have a backup of your data. In that case we'd say:

```
@private variable
```

Or the same thing but excluding the central Ether server, only your private ones:

```
@private.managed variable
```

There's also such a thing as default privacy policies, which you might set to 'managed', which allows any of your machines to edit each-other's states.

There's then also the 'confidential' modifier, which defaults to '@local' or '@private' or '@private.managed' depending on your default privacy policy. It would be '@local' by default.

```
@private.confidential variable
```

Just like operating systems, you can have different access levels for different types of operations: reading, writing, or executing on your local machine.

```
@public.read @public.execute API_METHOD
```

Unlike operating systems however, you can also specify what kind of write operations any node can perform. We might for instance only want to expose a +1 operation publicly:

```
@public.read NUMBER = 0
 @public.execute += (== 1)
```

Note that the write permission is just a wrapper for execute, namely:

```
 @public.write NUMBER = 0

 NUMBER = 0
  @public.execute =
```

Having covered access permissions, this brings me to the next section. How do you access external machines and their states?

Every variable... has a Location

Every variable is aware of its location in some other graph(s). (Of which equipped structure, like fields and methods are excluded.) This might be its location in a function, or a remote location where the variable is hosted. But it can also be an abstract location with yaw/pitch like a videogame, or even an abstract (sub)Graph where it is located without a specific point. Or it might be a function which describe a path of 'landmarks'. "Location" is just a modifier for some other structure.

The file system exposed to the runtime then has the same assumptions as we have for any variable: A single location can have multiple values, and at that location can always be defined a further structure of locations: By which I mean each file is both a directory and a file with values.

One of the goals with deeply integrating locations of variables in the language, is that we can abstract all networking away. So that only people working on optimization of the language, really need to deal with the details.

Just like access permissions, we can also use the syntax of getting characters for loading in packages.

```
< @ether/.ts
```

Or using other protocols than the ether's, you can load

```
< @"https://orbitmines.com"/package
```

Domain names like

```
@"orbitmines.com"
```

are also reserved for that domain, and will default to the ether's port 37839. They could be claimed by someone who owns that domain to map it to a character registered with the central Ether server.

Names can otherwise be registered by making the following network call, which would later be worked into the Ether's GUI.

```
@ether.@USERNAME = @me
```

In order to make this call to you first need to give the central Ether server permission to host variables for you. So that everything you mark as @private/.../@public gets routed through its servers to others. You do this by setting your status:

```
@me.status = Hosted
```

This would always go together with setting your network status as online, which is equivalent to opening a port on your machine for the central Ether server to communicate with. By default, you're set to 'Offline', and no network communication is allowed in your instance.

```
@me.status = Online & Hosted
```

Of course they can still directly call your instance if desired, but for that you need to give the central Ether permission to broadcast the IP associated with your current machine. (This way your registered username is associated with an IP address)

```
@me.status = Online & Broadcast
```

Or if they store your domain name or IP, they could of course also directly communicate with your instance. For which you would of course have to set your status to 'Online'.

Allowing the central Ether server to host variables for you, also assigns you a UUID. Which like all usernames is accessible through @.

It's also worth noting that version control will automatically load specific versions of methods/classes. Manually that would be

```
< @ether/package <&6ba7b810-9dad-11d1-80b4-00c04fd430c8>
```

You can this way also create structures which are sharded, or partially stored on other machines. We could for instance say

```
"A1", "A2" @ @me.managed, "A3" @ @ether, "A4" @ @"192.168.1.254"
```

If you would start iterating this structure, network calls would be made when necessary. Optimizations of this sort of network access would have to be more intelligent than just requesting a single Node when necessary. So you'll see that partially mirrored structures will be synced. Of course when the structures are sufficiently large you'll see actual sharding. (I'm thinking as an example of a sharded large game world.)

Probability

Since we're working with superposed values already, a natural expansion of the concept is to that of probabilities. Where by default, if we'd have some superposed values OR'ed, there's a uniform probability distribution. Or in other words.

```
x = "A" | "B"
```

Evaluated to a single value,

```
x#.random
```

would be 50% each. We can assign probabilities in two ways;

```
x: String? = 0.2("A") | 0.5("B")

x: String? =
  0.2 => "A"
  0.5 => "B"
```

Where any non-covered probabilities, default to None, if the variable is optional. (If the value is non-optional, all cases need to be covered)

Nested probabilities, will of course be supported.

```
0.5(0.5("A") | 0.5("B")) | "C"
```

Note that only OR's are evaluated this way, an AND, would not be effected, we can then of course, combine the two. The following for instance:

```
0.5("A") & 0.5(0.3("B") | 0.7("C"))
```

Would be two separate probabilities, both evaluated. One for A, and one for B or C.

Note that unless the probabilities are explicitly evaluated to a value, they stay around like any superposed variable.

```
n (== "A") => 1
n (== "B") => 2

n 0.3("A") | 0.7("B")
// 0.3(1) | 0.7(2)
```

We unlock some powerful capabilities like this, I could for instance create the following type:

```
Binary{-> .next == 0.5(?.random)}
```

Which is like saying: A randomly generated string of boolean values (possibly infinite) whose length is determined by randomly flipping a coin whether there's a next value.

This syntax requires some unpacking, first consider this:

```
-> .next
```

This is a constructor for a ray which recursively calls the method on an object until none is found. Typically, you'd be able to say things like

```
variable -> .parent
```

And then say things like

```
(variable -> .parent).last
```

to get the first parent in some tree.

Then there's

```
?.random
```

which takes the current type and generates a random instance of it.

To illustrate the power of this new abstraction, you'd then also be able to say something like:

```
Binary{-> .next == 0.5(?.random)}.length
```

Since .next generates randomly, the length is a variable (infinitely generating) probability distribution.

We could, instead of a Binary, also reference an Array with arbitrary objects:

```
Array{-> .next == 0.5(?.random)}
```

There is however, a slight problem with the following subexpression:

```
?.random
```

Because we currently point to an arbitrary Array, this is any Node. In other words, it's an infinitely generating object, we can't just uniformly pick a random element from an infinitely generating object. Which is why this statement would fail. Instead, you'd have to define some other way to generate random Nodes.

Equality/Equivalence

The default check for equality,

```
A == B
```

is actually a check corresponding to some equivalence graph. And is automatically implemented for all objects. This check will attempt to traverse the equivalence graph until either side is rewritten into the other (judgemental equality). And it will force both sides to be evaluated.

This default equivalence graph is using a casting function:

```
Node{== 1}
 as (== String) => "A"

1 == "A" // true
```

It's worth noting, that unless specified, object's don't track their uniqueness. In such a way that if the object has the same structure and values as the other, they will be considered the same.

So these two different constructors, will be considered the same.

```
Point(x: 0, y: 0) == Point(x: 0, y: 0)
```

You can override this functionality, by simply override the '==' method, or by explicitly specifying another equivalence graph, which accepts any kind of iterable:

```
A ==<in: -> convert(.)> B
```

If you want to check for equality and ignore the equivalence graph, you would pass an empty equivalence graph:

```
A ==<in: None> B
```

You might also want to check for equality up to some type, which would ignore anything not equivalent to that type:

```
class Example < Number
 field: String

Example(field: "A") = 1 ==<Number> Example(field: "B") = 1
```

There is one key importance when considering equality, the location of that variable is ignored. Because technically, the following two variables, are in different locations:

```
2 == 2
```

You can include the variables location in the check, which will require the exact same variable (which is a way to check for uniqueness, but you won't have access to an equivalence graph). Adding a third =, we check for the location too:

```
2 === 2 // false

var = 2
var === var // true
```

(Technically, the second thing there: "var === var", also talks about the variable in two different locations, but in order for equality to make sense at least that difference is ignored, and we defer to the variable's referenced location instead.)

An example where you would use this, is that when comparing two characters, they're considered the same whether they're hosted remote or not:

```
@me == @me @ @remote
```

But forcing location to be part of the equality check, it fails:

```
@me === @me @ @remote // false
```

Then there are three additional functions, the familiar type check:

```
A ==.instance_of B
```

The check for whether it's equal in structure (ignoring values). (isomorphic)

```
==.isomorphic
```

And the following check on whether it is a subgraph, or whether a pattern can be found inside another graph:

```
"12" ~= "0123"
```

You're also allowed to match to the boundaries:

```
"ABC" ~= ⊢"AB"
"ABC" ~= "BC"£
```

Functional Equivalence

One of the types of equality which is often unsupported, for good reasons, is functional equality/equivalence.

There is the mathematical extensional equality. Which simply states that every possible input, maps to the same output in both functions.

Then there's the mathematical intensional equality, which is essentially equality of source code.

```
(2 + 2)** == (2 + 2)**
```

But both concepts have three things which make them incomplete, which are: (1) "at which level of description?" or in other words "in which language?". If we're really ambitious we could even keep asking that question until we hit physics.

And (2) what about an equivalence graph of functions, across languages and implementations (which is obviously infinitely generating). To my knowledge, this sort of equivalence has not yet been implemented by any programming language.

Another (3)rd thing you might want to consider, is that a function is equivalent up to certain method calls, which aren't reduced as part of the control-flow/source code equality, but we just assume as "part of the output"; a certain "equality of effects".

Since a lot of functions accept infinitely generating objects, extensional equality is off the table as a default. Not to mention, that without access to the source code, there's no way of guaranteeing that just because two functions seem to be doing the same thing now, that they will keep doing so.

The best we can do is to partially answer that question: The only thing you know is that, historically, the two functions have behaved the same way; given some finite number of results.

Which is why

```
A ==.extensional B
```

only resolves quickly for small types. And typically, instead it is scheduled as a quest looking for partial answers; which are available through its variables. Instead of forcing an iteration through input/outputs. I'm also making the following

```
A ==.historical B
```

check, which is the same as extensionality, but only checks the function's usages within their lifetime up to when the check was called. Whenever A was called, B is also called to confirm they do the same thing. This is a way of using function extensionality which always resolves.

You could load historical values from some other session by labelling the function out-of-order and accessing it by filling its variables.

```
check.//Fill variables

A check\ ==.historical B
```

Now, the more interesting, and the default implementation of functional equivalence, is that of intensional equality.

Since we already assume we have access to the control-flow of functions, that control-flow is the language we use to compare functions. Including any functions we can't reduce, or which we marked as effects.

My thoughts currently are to have this default to the language we're targeting; or the lowest possible level we have access to. But there are also good arguments to be made for having it be the highest level of abstraction.

The tricky part of intensionality, is the equivalence graph we're defining. Which is simply something which will have to be continuously improved throughout the lifetime of the Ray programming language. Luckily we can use exactly the same code the compiler would use for optimizations and (trans/)compilation of equivalent code for this exact functionality.

We might want certain kinds of equivalences and not others, if we assume that certain equivalent code actually executes something different; our equivalence might be ignoring information we wouldn't want to ignore. As an example: CPU bugs, we might hit them depending on whether we include certain code snippets we'd consider as equivalent to some simpler code.

The actual details of this are still an open question to me.

Arc: 2026/Planned features

Quests

There's a hard problem which demands a practical solution. Namely that the halting of any part of any program is unknown [44]. And since we interpret any variable as an iterable structure, there's no way of knowing whether that structure is halting or not other than to start traversing the structure to determine whether it is. (Or the structure's abstract definition, which in turn, still is a structure we'll have to traverse)

Instead of ignoring this problem, it's a central theme to the Ray language. It's a rephrasing of the problem in terms of (1) how many resources you decide to dedicate to which problem and (2) how you deal with intermediate results/variables.

This problem is of course divided into two categories: the quests for our players, and NPCs. But the same approaches are used for both: A quest is a function. In the sense that a function follows a path of "things to do" - essentially a glorified TODO list. Like a function you define things to do in parallel, or in sequence.

These intermediate goals, or essentially subexpressions can be as vague as you can allow for. Even a very ill-defined set of goals, can still be interpreted as a function.

Where we interpret the return value of the function as the 'reward' for the quest. Which may in turn be new functions.

There are a few open questions relating to quests:

(1) How do you assign a difficulty rating to a quest, relative for the character completing it. For players, this would be the difficult project of rating based on their current knowledge, whether some challenge is appropriate for them. Similarly for NPCs, if we mark their access to hardware resources, combined with their capabilities, how difficult is it to execute some function?

(2) How does the reward of the quest influence our future? Some sort of effectiveness rating? This could be measured in the sense of, which other functions/quests does the reward effect: Their difficulty rating, and the compounded effect of their future rewards. Which is pretty similar for both players and NPCs. The only problem being that for players, quests might be much harder to abstractly verify that they meet your required conditions for completing a quest; and then their rewards.

(3) For NPCs, Quest selection, or what to spend resources on.

Throughout our code, we might have many checks which require on some possibly infinitely generating traversal of a graph:

```
if graph.last
```

We might say assume it halts, and rollback any effects (if possible), when it finally terminates.

```
if assume graph.last
```

Where it executes with the assumption, and then spawns a quest for the NPC to resolve this possibly infinitely generating evaluation.

Speculatively assuming other cases would be done the same way:

```
if graph.last
 A()
else assume
 B()
```

Then there are different execution modes for an NPC, keep running until all quests are resolved, keep running after, waiting for more. Or stop running with unfinished quests. Where any errors thrown could also be interpreted as unfinished quests for fixes (which the character monitoring it, might pick up)

(4) For gamification of science, engineering and education. A big open question is how do you generate/find quests relevant relative to some model one has of a player's knowledge. Of course one could have a curated library of quests, but generation of new ones, seems a crucial capability. Which is probably best suited to be paired for some way of navigating/finding them - in game form.

I'm hoping to explore answers to these questions over the coming years.

Templating other languages

A perfect example of where text-based programming languages fall short is the templating of other languages. The simplest example of this is how (templated) strings work. Whether it is to support expressions within a string, or to have to escape the character we're using to close the string's expression. This necessity of having to escape the characters, is the shortcoming.

If we were to step away from text-based programming languages, our editor could simply be aware of whether we're working in the language we're templating or if we're defining an expression in our host language.

This is then also one of the features our editor The Ether will have, alas, we're currently defining a text-based language, and we'd still like to template other languages. Namely because we're going to be implementing (trans-/)compilers to other languages within the language.

I've resorted to using the double parenthesis unicode characters for injecting expressions in the Ray language, as I don't think any of the languages I've looked at the past few years uses them.

```
x = 2
program = ⟦ts
 const x: number = ⟦x⟧ * 2

program().x // 4
```

Besides simple templating, I'm hoping to also allow bindings such that all the functionality of the Ray language can also be injected into arbitrary code from other languages. So if you want access to the function control-flow or intermediate values of variables, that would be supported.

Another thing that I'm hoping to support is to, instead of running the language in its native runtime, to extract the control-flow (which could even be one which doesn't support the entire language), and run it like any other program in the Ray language.

By the end of 2026, I'm hoping to have something more definitive regarding this feature.

Other planned features

There are other features I'm thinking about which I'm not yet sure about, so design decisions on it are still pending.

(1) Whether to superpose concurrent accesses of a variable. If we define 'branch' as separate execution branches of a function, so there's concurrent access to some variable:

```
x = 0

branch
 x = 1
branch
```

```
x = 2

x // What is 'x' here.
```

We could have the compiler realize there's a concurrent access, and instead of forcing some race condition, we could simply have it superpose the different branches' variable changes. And say

```
x == 1 | 2
```

(2) Is a function's control-graph without an initial boundary: a starting point, a valid function? So that would be one which has an infinite past; one could imagine that we could still deduce certain things about function execution. Things like imagining starting at any point within some looped initial structure. Or properties and possible values certain variables can have based, even on an infinite past.

(3) Kind of related to that, is starting with certain variables down in a function, and running a function backwards, using isomorphisms (defined or automatic) or historical values where available. And of course how you'd define and call and define inverses of functions. I've not yet thought through how that would work properly.

(4) Eventually once we have a programming language which isn't just text-based, you can imagine monkey-patching an existing function or constructor; by simply editing its code. Is there some way in which we want to implement this in the text-based form too?

(5) One of the open challenges is how you would create a massive decentralized graph structure which has dynamics defined on it; across shards and redundancies. The basic infrastructure for it is there, but how you would do this optimally with, as a challenge for scale, a million players simultaneously.

(6) One thing I want to research is to limit some speed at which one can explore a game graph in some way such that the player's speed is actually limited by some computational restraints. In such a way that you can't just skip ahead programmatically. One immediate possibility which jumps to mind is that you need some verifiable history of your location. But how to combine that with a specific world graph, and how to make it distributed, I don't know yet.

(7) The intuition I currently have for implementing a GUI, is that it serves the language best when speaking about it geometrically. When defining what mirroring features of languages like HTML or other GUI libraries in other languages do, being able to compare what they do geometrically seems like the right intuition. A big part of eventually becoming a rendering engine, also will rely on libraries like this.

Currently, you would implement these structures similar as you would higher inductive types. By simply structurally describing them. An example would be a loop, and its 2d interpretation, a circle, whose definition would be at least something like:

```
Point = Ray
Loop = Array.Unbounded.loop

class Circle
 outline: Loop{map(to centre -- #.min.length).reduce(==)}

 centre: Point
 radius: outline to centre -- #.min.length
```

One can imagine defining, other shapes, functions like centering, to similarly follow a more intuitive definition. And have less-intuitive shader code be generated from this sort of high-level representation.

A big geometric library is therefore on the agenda when I'm going to work on the IDE.

Wrapping up

Having skipped the 2024 progress update due to personal reasons, I hope to continue this yearly exposition of thoughts and ideas I've been working on. From now on they should be much more technical.

My current timeline

My current predicted timeline is as following:

- 2026: Have a functioning runtime ; just a v0 specification runtime, without any optimizations which I can run in the browser. At the end of the year I'm hoping to make a language specification book which I'll update periodically.

- 2027: Start work on the IDE: Ether, and start lifting the Ray's language constraints of being text-based

- 2028: Using the IDE, create functionality for the analysis, indexing and implementation in/of other programming languages: A start to the library project.

- 2029: Think about turning the Ether into a fully-fledged rendering engine for 3d-environments/physics/games.

- 2030-2035 Start research on gamification of science, engineering & (technical) education

Join Our Discord to be notified about progress updates.

Footnotes & References

[1] Fadi Shawki. "On the Intelligibility of (dynamic) Systems and Conceptual Uncertainty" OrbitMines Research (2022)

[2] Fadi Shawki. "On Orbits, Equivalence and Inconsistencies" (2023)

[3] Fadi Shawki. "OrbitMines as a Game Project" (2024)

[44] "Halting problem"