Roc Editor Protocol Proposal

Motivation

Historically editors were built to emulate editing of the paper script in the constrained environment of computer terminals. Paper scripts with written programs are fed to the script processor, in modern lingo, programing compilers, which would prepare a set of machine instructions, and we would get a form of machine that was programmed a certain way.

REPLs work very much the same way, we feed smaller programs, they got read, evaluated and the result is printed. You feel how much the same vibe of papers being read is still present.

If we want to provide a somewhat different experience of programming computing machines, the hypothesis is, we should have a better interface to communicate modifications to the compiler, so we can have a more immediate way of seeing and modifying the logical structure of our programs and inspecting the way how data flows through our program.

Main concepts

The structure of our software is directed graph, or non-circular tree as I like to think about it. Everything is "just data", bits layed out in memory, nobody is claiming dualism here, but the difference between structure of the program and data flowing in-and-out is that the "program" is a tree that is not supposed to change over time, while obviously data flowing through the branches is orthogonal to it, and it is supposed to change over time. Hence we capture data-flow for a reason. That is a Heraclian data river that get perturbation just by flowing through Plato's crystalline structures.

Rough idea is that Roc compiler is the software that is continuously running, pretty much acting as a server that, is tracking and modifying both the crystalline structure and the orthogonal viscose goo that gets morphed once gets input trough goo-morpher made out of crystals. Dear reader, I will soon drop this poetic jargon, but I would like to build an intuition about how we might treat software development in this protocol proposal.

Protocol itself would consist in communicating with the server about the surroundings of the cursor, our sole focus, and capabilities that the user might have to effectively modify that crystalline structure. I am not sure if commiting to the wording of AST for the crystal structure would be correct, because I don't possess enough knowledge in the workings of the Roc compiler. My intention is to capture the logical structure of the program from the user's perspective. In it we would not have the concept of files nor modules, unless we introduce OCaml style polymorphism which would break this conception. Each node on the graph would be designated with a path. For any position of the cursor the server would inform us what we could do next, in the sense of modifying, or testing parts of the structure and what are the problems with the current structure.

Pointing to parts of the structure with paths

The path of the graph is insensitive to the correctness of any kind, except that there are no stay nodes, which would equate to stay characters in standard code editors.

Let me try to illustrate that with simple example, take this today valid Roc code:

```[tui.roc]

app "tui"
	packages { pf: "tui-platform/main.roc" }
	imports [pf.Program.{ Program }]
	provides [main] { Model } to pf

Model : Str

main : Program Model
main = {
	init: \{} -> Str.join " " [ "Hello", "World"],
	update: \model, new -> Str.concat model (Str.repeat 3 new),
	view: \model -> Str.concat model "!",
}

```

Starting block is the zero-node which is not an access point of a program, but a point from which all paths lead from.

To reference the whole `main` function we would say `tui/"main"`, not very exciting. But to point to `Model` part of the type signature we would say `tui/"main"/type/0/0`, for `model` used as argument to the `Str.concat` in the `view` function we would say `tui/"main"/expr/rec/"view"/expr/fn/1`.

As you can tell few things are worth explaining. I stuggle with proper formal names, and formulating proper grammar, because I did rolleyes and slept on my logic classes when we were talking about making notations, please bare with me if something is wonky or too simplisitic.

There are two essential kinds of things that Roc program contains:
- data types
- functions

I would argue that modules are non-essential in the sense that they are needed to separate pieces of code trough files. If they are non-essential then `imports` and `exports` parts are also non essential. The argument that in this would break the idea of modularity, is not true, because modules are just bags of essential things with complex names. If we start treating modules in Roc as OCaml does it, this wouldn't hold true, but I try to force simplicity. I believe that on the level of the editor client it would be possible to formulate rules about ability to reference certain nodes along the graph, which would effectively simulate modules and hidden functions. But they are non-essential.

Platforms are demi-essential: if our code is an app then they are essential, so far that we would have to choose one, and attach our code to already exising code structure. Same goes if we want to build a library for a specific platform. But if we are building a general purpose library they are irelevant.

Backpassing syntax sugar that is present in Roc is non-essential: if we have control over the presentation of the code, showing lambda's with inverted arrows becomes a matter of controlling how we would render it.

Functions

As I could understand there are few ingredients to a Roc expression:
- Name
- Type signature `type`
- Parameters `params`
- Values `vals`
- Return expression. `expr`

Name of the function is distinguished from the name of the type by being starting with lowercase letter, from there you have different branches with zero or more members.

Type signature

Since any type signature is basically a list of types where the last element is return type it makes sense to write them numerically. Since on any position you can have a type with any number of type variables subsequent it makes sense to refer to each of them as the next level of nested indices.

So in a made up type signature
```
a, Str, Foo z g q -> Triple U64 Dec a
```

`Foo` could be pointed by `type/2/0`,
`q` could be pointed by `type/2/3`
and entire `Foo z g q` could be pointed by `2`

Params

Parameters specified after ` = \` also behave as a list of this with the caveat that nesting indices would refer to parameter deconstruction:

```
 = \ a, str, (Foo zoo goo qux) ->
```

were to `a` would be `params/0`, and `qux` would be `params/2/3` .

Values

If a function contains intermediate values they need to be referenced by name since they are order independent, hence, my hunch is that we would need to refer to each declared value by its name, hence:

```
 \ a, str, foo ->
   zoo = calcualteZoo a qux
   qux = getQux foo
   ....
```

```

If you would point to the whole `qux = getQux foo` it would be `vals/"qux"` while the reference to the `foo` part would be `vals/"qux"/expr/1`

#### Expression

Or that-what-is-returned-from-a-function, please feel free to correct me on the naming. There are few things that can be returned:

- numbers, string or a tag `lit`
- a record `rec`
- a list `list`
- function calls  `call`
- lambda `fn`
- pattern matching `when`
- Pipelines `pipe`

Going back to our first example, lets break down different forms of exprs

```
main : Program Model
main = {
        init: \{} -> Str.join " " ["Hello", "World"],
        update: \model, new -> Str.concat model (Str.repeat 3 new),
        view: \model -> Str.concat model "!",
}
```

##### Records

The content of `main` is basically a `rec` where each field is defined as a value of the similar form.

To refer to whole record literal we would use `main/expr/rec` and each field could be referred in the form `main/expr/rec/init`. The content of each field is defined in the same manner as a function call.

##### List

If we would return a list of stuff aka
```
\{} -> [1, 23, countYs "Yeah!"]
```

Then we would refer to a list with `.../expr/list` and each element would be index-referred so: `.../expr/list/1` would refer to literal 23. At path `.../list/2` we would have a function call, so to refer to literal "Yeah!" we would say `.../list/2/call/1` .

##### Lamda

If we have following code
```

``` { update = \model, new -> Str.concat model new } ```

The value of the `update` field is a function without name nor type definition. Lambdas hence are poor function definitions. To point for example at the whole `Str.concat model new` thingy we would have a path of `.../rec/"update"/fn/expr`

Function Call

From code editing perspective each function call consists of the name of the function to be called at first position and arguments on each subsequent slot. That means that the call could refer to each element by the index.

```
        launchMissles "ACME" Coyote [Beep, Beep]
call  0           1        2         3
```

Pattern matching

A situation with pattern matching makes things a bit trickier, let me illustrate with example
```

Tags a : [Very , Much U32 , Uber (Tags a) , ForgetIt]

\t ->
        when acqurieTags t is
        Very -> makeCookies
        Much x -> positionCloud x
        Uber ((Much x)) -> repeat x
        _ -> nameThatCanBeNamedIsNotEternalName

```

to point to the whole pattern matching business we would say simply
`.../expr/when` because the expression captures the whole thing. If we would like to point to the `acquireTags x` part we would reference it with `.../expr/when/call` and each individual part would be pointed similar as we are doing regular function call, with numbered indices: `.../expr/when/call/0`

The part of referencing each pattern branch consists of elements that are pretty similar to the lambda definition, where each pattern is number indexed, and then follows lambda form: to point to the `Very -> ...`
we would say `.../expr/when/pattern/0` and if we would like to reference the `Much x -> ....` we would go with `.../expr/when/pattern/1/params/0`. Similarly if we would get inside the patterns branch `positionCloud x` we would say: `.../expr/when/pattern/1/expr/call`

##### Piping
[tbw]
### Data type definition
```

When we are defining types in Roc we would do that on the working surface of the module. Types are distinguished from the function calls with the fact that their names start with capital letter.

```

Person : {
	name : Str ,
	age : U8 ,
	}

Moods :
	[ Happy,
	Grumpy,
	Bympy,
	]

Woods forest : Woods (List forest) Moods

```

If we say that we would point to any of those type definitions as a whole we would point to it as `.../Person` , `module_name/Moods` .

Similar to the function definitions there are few components to a type definition:
- type parameters `params`
- type expression `expr`

Following it, the types themselves can be either

To point to `forest` as defined as type parameter we would do very similar thing we would do to a function definition: `module_name/Woods/params/0`

Types of values at end of the path

Each path given back to a server has to be checked for its validity: it is a valid path that would point to a certain thing. That thing might have a textual value, which doesn't have to be a valid identifier in the Roc program. Also that thing might give more information about what kind of thing sits at a given path.

Interacting with server

- incremental graph reveal

- extending graph, making stuff
- trimming, deleting stuff
- plucking, moving stuff around