Thesis: message formats are not inherently complex. Complex types exist in the system, but not at the level of message processing; handling them is a delegated behavior. However, we have a few ways of representing them internally; they have different purposes (e.g. Protobuf for “your language already has a compiler for protobufs” interoperability, Erlang terms for use inside the node and between nodes on the same VM host, Juvix and noun for execution) but they express the same things.

Therefore, I present an ur-format with very few moving parts for defining messages, with a sort of pseudocode-ish representation, along with ways to represent it in the various languages present in the system. It’s meant as a common denominator, not necessarily a lowest one but a low one.

Opaque

We start by introducing an “opaque”. This is just something which is irreducible as far as the message definition is concerned; it could be anything, but it’s not the message format’s responsibility to look inside it. (It might compare it for equality, sometimes, but I see that as pretty extensional; in cases where it might be problematic, like a program, this is already “a Nockma program” specifically and noun equality suffices.)

Some examples of opaque things could be:

- an integer
- a JPEG of a cat
- a program
- a proof
- a message that we’re not processing here and now

In our pseudocode, this is just represented descriptively.

integral-message : integer

We handle an integral-message by making a ‘ping’ sound if it is nonnegative, and a ‘pong’ sound if it is negative.

cat-message : jpeg

We handle a cat-message by decoding the JPEG and displaying it on the screen.

We could just say “binary” here and not be far off, but saying that would imply some sort of requirement to marshal them to bits, always, which is often redundant. Say it’s a function; we would really prefer to just pass it around, until we reach a boundary where that’s not possible, because we probably plan to evaluate it soon.

However, we do want our opaque things to be able

to be marshalled to bits; we might send them over the network. If you can make it a noun, you get jam for free; this works for e.g. Nockma programs. If it’s already a binary, you’re also set; this works for e.g. Cairo programs.

Record

The above doesn’t suffice for most messages, which have multiple fields. We therefore introduce records with named fields. This is reminiscent of a product type, but the fields do have names; this is type information about them, so you could just say it is a product type and have a strong case.

Order matters here sometimes, but it might not matter in all particular language representations. If there is an order, the one given is canonical, else it might just be a map from field name to field value.

In our pseudocode, we’ll use square brackets:

cat-message : [name: string id: natural image: jpeg destination: node-id]

In Elixir, this would look like (using the typedstruct macro to shorten things):

typedstruct do field(:name, String.t()) field(:id, non_neg_integer()) field(:image, JPEG.t()) field(:destination, Anoma.Node.node_id()) end

In Juvix:

type CatMessage := catMessage@{ name : String; id : Nat; image : JPEG; destination : NodeID; };

As a protobuf, our opaque things are nearly always going to be bytes:

message CatMessage { string name = 1; bytes id = 2; # if it's capped, could be uint64 or similar bytes image = 3; NodeID destination = 4; }

As Nock the field names are type information, so it would just be a cell (following the brackets-associate-right rule to have tuples larger than 2). Giving some pseudo-Hoon since our Elixir compiler syntax isn't decided yet:

+$ cat-message $: name=@t id=@u image=@ destination=node-id ==

Whatever the compiler, it's capable of knowing that name is at 2, id is at 6, image is at 14, and destination is at 30; this is most of its job.

Tagged union

Tagged unions are not sum types, but we want something sum-type-like and only Elixir and Hoon support them. This isn't a huge deal since most practical sums include a tag anyway, but it will matter for some representations later.

We use the vertical bar in our pseudocode, since this is the most common syntax, and angle brackets to set them off since these are the least commonly used brackets.

cat-or-vampire-message : [name: string id: natural image: | destination: node-id]

(Vampires, of course, can't be photographed.)

Elixir has sum types, so it can be simplified somewhat to:

typedstruct do field(:name, String.t()) field(:id, non_neg_integer()) field(:image, JPEG.t() | nil) field(:destination, Anoma.Node.node_id()) end

However, most cases will actually need the tag; in practice it'll look more like {:cat, JPEG.t()} | :vampire

({:vampire, nil}

would be very non-idiomatic redundancy).

I am not actually certain how to do this in Juvix without pushing the tag outward to the whole message; I just don't know enough Juvix. I think you'd define something like

type CatOrVampireImage := | Cat JPEG | Vampire ();

and use it as the field, possibly.

Protobuf would use optional

because this is an option:

message CatOrVampireMessage { string name = 1; bytes id = 2; optional bytes cat_image = 3; NodeID destination = 4; }

However, for something more complicated than "optional" it would use oneof

instead. This comes with the tag for free (you can check which field of a oneof

is set); it also comes with some concerns about ensuring the field number is unique, but all protobuf has that.

For nouns, we're going to need the tag because atoms are our only ur-element so we can't just sum them with nil or similar; we use 0 for "nil". Pseudo-hoon:

+$ cat-or-vampire-message $: name=@t id=@u $= image $% [%cat jpeg=@] [%vampire ~] == destination=node-id ==