Meta-execution

Paul Christiano Follow
Oct 26, 2016 · 6 min read

This post describes meta-execution, my current proposal for capability amplification and security amplification.

(*Meta-execution is annotated functional programming + strong HCH + a level of indirection. It is implemented in the amplify module of my ALBA repository.*)

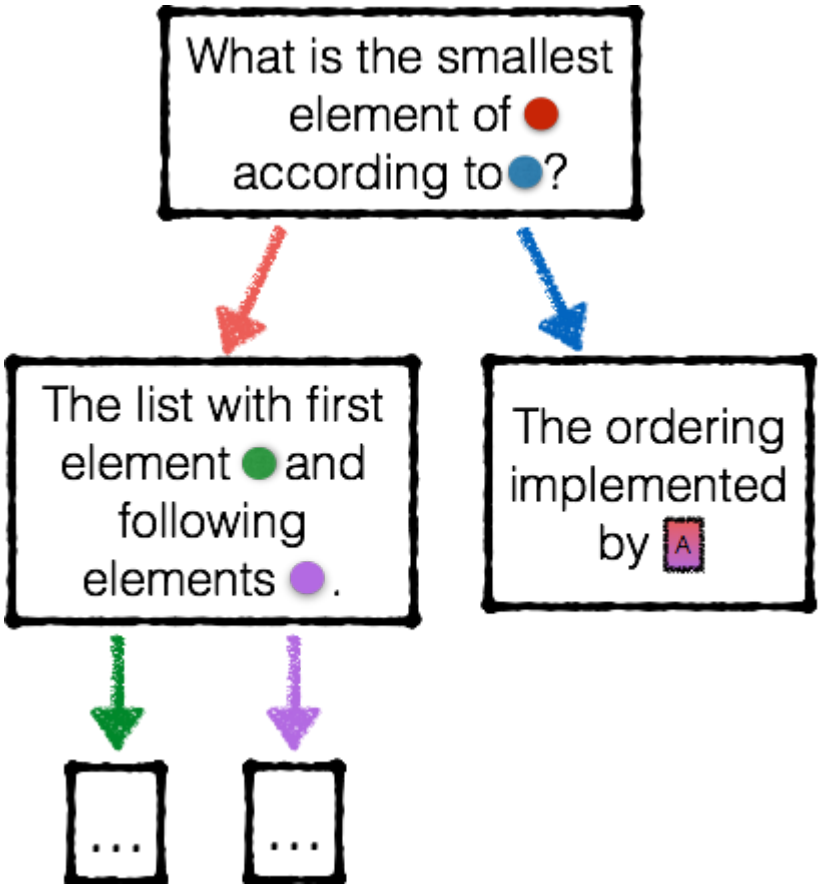
Goal

We are given an efficient agent **A** which competently pursues some values. We'd like to use a bunch of copies of **A** in order to implement a more powerful and robust agent Meta(**A**) with the same values.

Outline

Our basic plan is to build a machine out of copies of the agent; instead of asking the agent to make a decision directly, we ask it to implement the decision-making process by answering a sequence of questions of the form "what should happen next?"

The basic object in meta-execution is a **message**, which consists of text along with pointers to other messages or to agents.

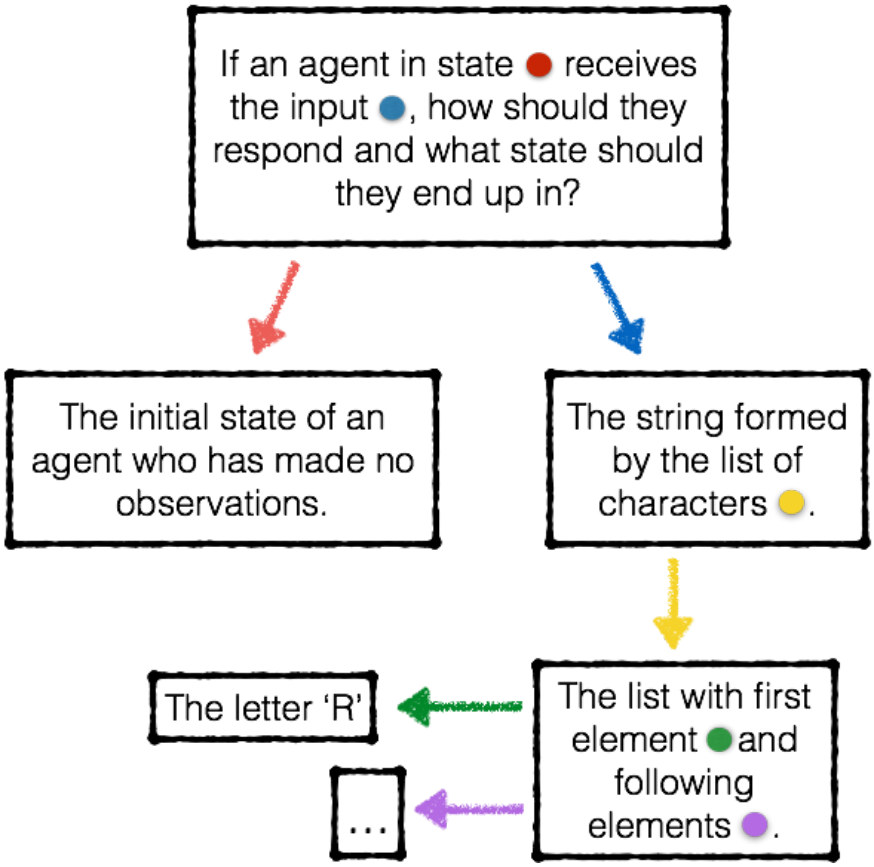


Each box is a message. **A** is an agent who can respond to queries like "which of X and Y is larger?"

We can represent arbitrarily large objects as giant trees of messages and agents.

Meta-execution first forms a tree representing the question "what should be done?" It then asks the agent **A** to perform a sequence of operations on the tree that eventually lead to an answer. Then it executes that answer.

The initial tree might look something like this:



If you can answer this question, you can implement an agent.

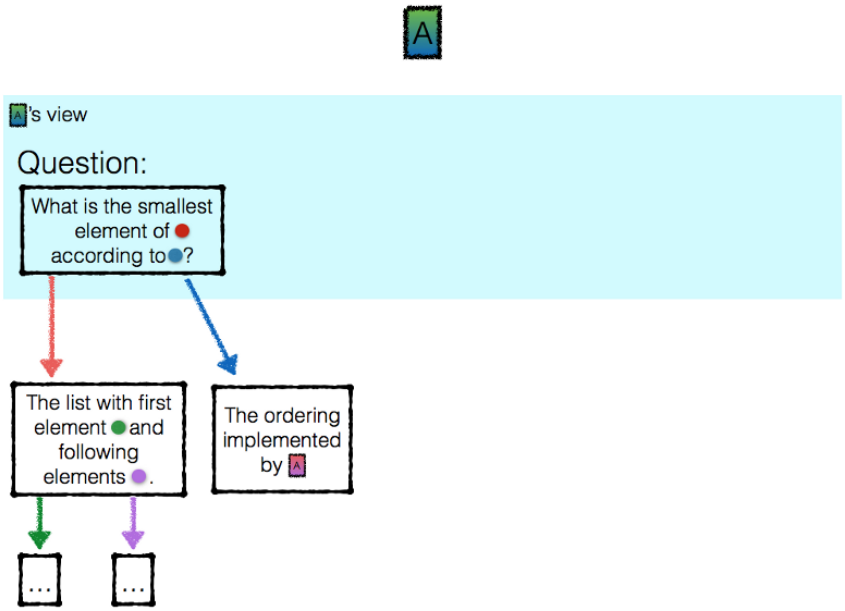
At any given time, an agent who is operating on this tree can only "see" a few messages: it can read the text of those messages, and see pointers like [red] and [blue]. Initially the agent can see only the root of the tree.

If you are an agent tasked with processing a message, there are a few basic operations you can perform. You specify the "targets" of the action by specifying pointers you want to follow:

- Look at another part of the tree which is not currently visible.
- Spawn a new agent, and see a pointer to that agent.
- Send a message to an agent, and see its reply. You can compose a message by writing it out with sub-messages in parentheses (); for example, "What is the smallest element in (the list with first element [green] and remaining elements [purple]) according to [blue]" would produce the message in the first image above, if [green], [purple], [blue] had appropriate values.
- Terminate the current computation by composing a reply. This reply gets sent to the "parent" who initiated the current computation. In the case of the very first agent, who was created

> in order to answer the original question "what should an agent in state [red] do after receiving input [blue]?", the reply specifies what the overall system should do.

An example execution is illustrated here.



I made a quick demo of the execution process, you can find it here.

And that's basically it. We spawn a new agent, and hand it the "what should we do?" message. It can then take any of the basic actions listed above and see the result. We repeat that process until the agent returns a message indicating what should be done. We parse the message as an action and new state (see the section on parsing below), we execute the action, and we update the system's state.

The details

Hopefully for most purposes that outline tells you everything you need to know. If not, the easiest way to learn *exactly* how this works is probably just to look at the code.

Meta-execution is implemented as `lambda A : Meta(HCH(A, n))` in the package amplify.__init__, where `n` is the computational budget and `A` is the meta-executor. You can experience *being* the meta-executor by calling `examples.meta.act("test")`. The available commands are described in the README.

Everything is immutable

I assume that we have a digital implementation of **A**, and so we can snapshot and copy it freely. Each time an agent makes an observation and then takes an action, we keep around the original copy of the agent as well as the new copy.

For example, suppose that I send **A** the question q, and she replies with the answer a. The original agent **A** will not be changed, but I will get back a pointer to a new agent **A′**. There are now two different agents to whom I could send my next question. **A′** remembers seeing the question q and replying with a; **A** does not.

Both messages and agents are immutable.

You can't modify a tree of messages, you can only compute a new tree which happens to share some subtrees. If I send a message like "sort the list [purple]," I should expect to get back a reply like "the sorted version is [black]." The original pointer [purple] will never change. This should be familiar to anyone who has experience with functional programming, but might be a little weird to people who aren't used to it.

Parsing outputs

After we run our agent on the "what should we do?" tree, we obtain a message. We need to parse this message as an action. We can do this with the help of an agent, but it requires asking somewhat more specific questions.

First we ask "What string should the agent output?", and then only after we ask "And what state should the agent end up in?"

In response to "what string should the agent output?" we receive a message. To parse it as a string, we can ask "What is the first character of the string?" and then "What is the string obtained by removing the first character?" To parse a character we can ask "What is the ASCII code of the character?" and get a message. To parse an integer we can ask questions like "Is the integer zero?" and then "Is the integer even or odd?"

Everything can bottom out with yes/no questions, to which the response ought to be either the message "yes" or the message "no." If it's not, then we can ask again. (The question can make it clear to the answer that the response ought to be yes or no.)

The answer to "what state should the agent end up in" is simpler—the state of the agent is itself a message, so we can just take this at face value.

Budgeting

We can modify the scheme to include a computational budget.

When we create the first agent, we assign it some initial budget, like 10^9 operations.

Each time an agent views an object or sends a message, it uses up 1 operation from its computational budget. When an agent runs out of computational budget, it must immediately compose a reply, it can't do any other operations. (It is free to return an error message requesting additional resources and explaining why it ran over budget.)

Moreover, each time any agent sends a message and expects a reply, it has to also transfer some of its computational budget to the recipient. The recipient may then spend that budget in order to come up with its answer. When the recipient responds, it may return whatever computational budget remains unused.

Why be functional?

Meta-execution as I've described it is purely functional, and the computational model very closely resembles Lisp.

At face value it looks like there is a huge range of universal models that would all serve equally well.

But once we are interested in alignment-preserving computation, I think there isn't that much room to vary this scheme without making things much uglier. If anyone feels like a different approach to meta-execution would be comparably natural I would be curious to hear about it.

The basic problem is that it's completely unclear when you "should" mutate part of a mutable program's state. Every imperative program has a different set of contracts and expectations for how different objects or parts of memory will be used. For meta-execution, any such expectations need to be baked into the agent that we start with. That means that we are stuck with whatever we have at the beginning, and so in some sense we need to start with a "universal" contract.

We have an implicit notion of “accurate” and “helpful” for questions and dialogs. For the version of meta-execution I’ve described, that’s all you need—you never mutate an object, and so you never need to know how an object is used, you just need to figure out what message you should return in response to a question. I don’t see any comparably simple approach in any other computational model.