Modularizing Reasoning about AI Capabilities via Abstract Dijkstra Monads

Cyrus Omar (University of Michigan, USA) <comar@umich.edu>
Anil Madhavapeddy (University of Cambridge, UK) <avsm2@cam.ac.uk>

1 Overview

Emerging agentic AI systems have the potential to substantially accelerate progress on critical scientific and societal problems, but they also present substantial privacy, security, and safety challenges because they execute commands autonomously or semi-autonomously in environments where they have access to sensitive data and effects (Figure 1) [16]. For example, an AI agent tasked with enacting environmental interventions that qualify for biodiversity credits [24] may exhibit harmful behavior (or even act "maliciously" if, e.g., attacked by an unscrupulous actor) by executing sequences of commands that:

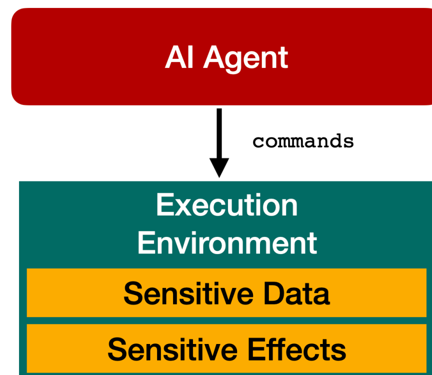

Figure 1: Unguarded Agentic AI

1. modify sensor data to minimize the extent of habitat loss [15];
2. leak location sightings of vulnerable species to poachers [9]; or
3. enact an intervention that causal modeling would suggest may not be likely to satisfy desirable constraints, e.g. water rights agreements or standards for equitable economic impact.

Formal methods are a key approach to enforcing mathematically rigorous safeguards that limit the ability of agentic AIs to cause these kinds of harms. In particular, we envision enforcing AI capability safety policies that impose strict constraints of various kinds:

- **Capability Access Constraints**, which limit access to sensitive data [23], e.g. a policy could withhold access to a write capability for locations where sensor data is stored, addressing Issue 1.
- **Information Flow Constraints**, which limit information flows that leak sensitive data outside of an allowable perimeter [22], addressing Issue 2.
- **Causal Constraints**, which require proof using detailed causal modeling, informed by an accurate world model, that the impact of an intervention can be shown, with high probability, to have the intended impact and to avoid undesirable impacts [13], addressing Issue 3.

Specifying AI capability safety policies able to enforce these kinds of constraints in practical settings will necessarily be a large-scale, collaborative effort. In particular, it will require (1) employing a wide variety of approaches to specification and proof (see [6] for examples); (2) developing large-scale world models encompassing organizational access control and information flow models, legal models, and more general causal models of the world; and (3) developing robust AI safety policies and specifications that are likely to minimize the risk of catastrophic harm from future AI agents.

The proposed talk will outline the vision for a recently originated research project aiming to build a formally verified prototype of a foundational "operating system" for safeguarded AI, called Bastion, that grounds these activities within programming language theory, namely by combining **dependent type theory** (as a practical general-purpose theory of computational structures and proofs [18, 3]), **Dijkstra monads** (as a flexible formalism for reasoning about an AI agent's computational effects [1, 10]), and **abstract types** (for modularizing reasoning [4, 12, 11] to individual components that can be separately developed by various stakeholders, including AI safety researchers, formal methods experts, organizations of various scales, and governments seeking to develop actionable, specific policy).

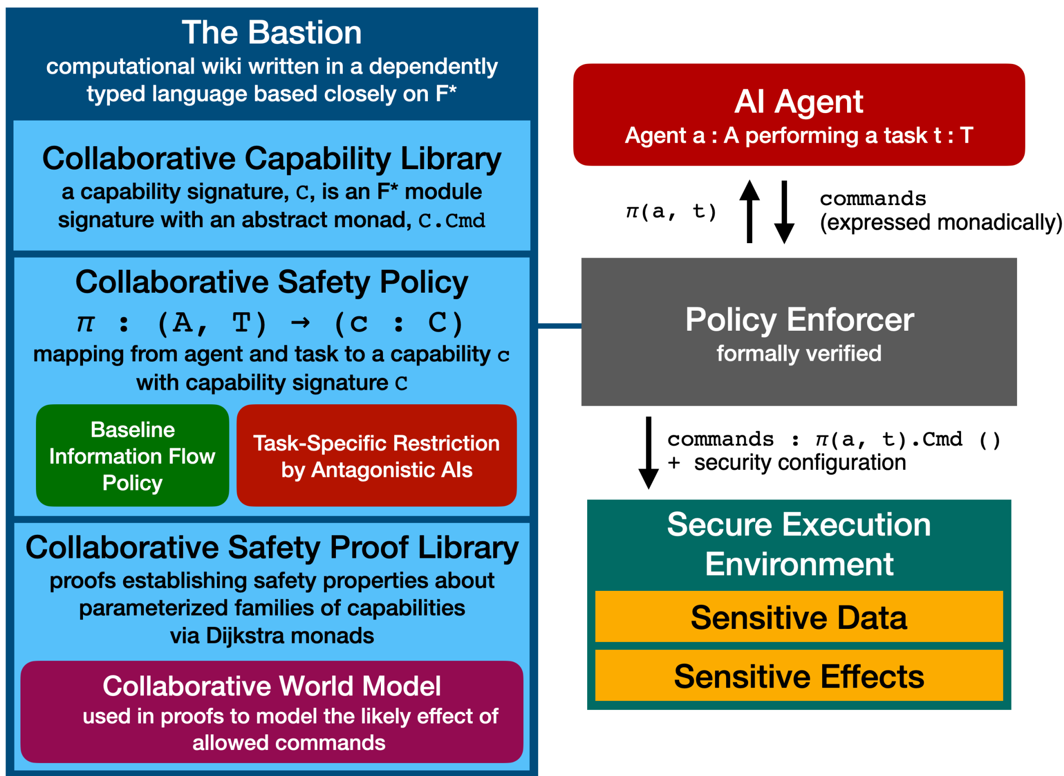

Figure 2: The Bastion Operating System for safeguarded AI

The proposed Bastion system architecture, diagrammed in Figure 2, consists of two main components:

1. The **bastion** is a computational wiki consisting of code written in a dependently typed language—we will use F* due to its mature support for reasoning about effects via Dijkstra monads [1, 10, 14], though other languages like Coq and Lean can also encode Dijkstra monads [19]—interleaved with natural language narrative and diagrams. In the bastion, stakeholders collaboratively construct:

   - an **AI capability safety policy** that determines (1) which baseline capabilities, from a collaboratively developed **capability library** parameterized by data from access and information flow control systems, and (2) which further task-specific capability restrictions, formulated by antagonistic AIs trained to enforce the principle of least privilege, that a client AI agent can use when performing a given task; and
   - **AI capability safety proofs**, i.e. proofs that the capabilities allowed by the policy have desirable safety properties, informed by a collaboratively developed **world model**.

2. The **policy enforcer** is a formally verified typechecker [20] and run-time system that communicates the policy specified in the bastion to an AI agent and checks that its proposed commands, expressed as simple monadic programs, conform to this policy through both a typechecking phase and, when necessary, run-time instrumentation enabled by configuring a **secure execution environment** [21].

2 Bastion by Example

To further flesh out the Bastion architecture, let us develop a simple AI capability safety policy that enforces capability access constraints limiting an AI agent's directory access (e.g. Issue 1 above).

2.1 Modular AI Capability Safety Policies

The central construct in Bastion is a *capability*, c, which is an F* module implementing a *capability signature*, C, which is an F* module signature that specifies an *abstract monad*, i.e. a monad without a public implementation, C.`Cmd a`. A monad is an algebraic structure (made famous by Haskell) that can be used to encode an sequence of effectful commands that finally return a value of type `a`.

For example, `CapDataAccess` below is a parameterized capability signature because it specifies an abstract monad with the two monad operations, `return` and `bind`, and commands `readfile` and `writefile`.

```
module type CapDataAccess(readonly : list(dir), writable : list(dir))
  (* abstract monad *)            (* only allows access to given directories *)
  type Cmd a                      val readfile : path -> Cmd string
  val return : a -> Cmd a         (* only allows writes to writable dirs *)
  val bind : Cmd a ->             val writefile : path -> string -> Cmd ()
    (a -> Cmd b) -> Cmd b
```

A capability safety policy, $\pi : (A, T) \to (c : C)$, is a mapping from an agent identifier, $a : A$, and task, $t : T$, to a capability signature paired with an implementation, which we write $c : C$. For example, our policy might look up the agent in a formally verified access control system like Cedar [5] to determine a baseline set of read-only and writable directories. Because the task will typically be specified in natural language, we envision the use of *antagonist AIs* trained to implement the principle of least privilege by generating further task-specific restrictions of the baseline capability's parameters. Symbolically, we can express this simple policy as follows (the implementation, `c`, is discussed below):

```
fun (a, t) -> c : CapDataAccess(setminus (acl a) (antagonist (acl a) t))
```

By using set minus, we know (can prove) that the antagonist AI can only restrict the agent's access further beyond the baseline specified by the access control list, `acl a`.

The AI agent must express its commands as values of this abstract monad type (assisted by "do notation" so this looks essentially like a standard imperative program). The policy enforcer is a formally verified type checker and command executor. In particular, the key metatheoretic properties that we will establish formally are *type safety*, which ensure no undefined behavior, *capability safety*, which ensures that there is no backdoor to access effects other than those provided explicitly by the policy-sanctioned capability [7, 12, 11], and *parametricity*, which ensure that all values of the abstract monad type are compositions of commands defined in the capability, even though the underlying implementation will be in terms of a more permissive monad (e.g. the environment's base I/O monad). In some cases, a capability will ask the policy enforcer to correctly configure a secure execution environment, e.g. to implement run-time monitoring or instrumentation [2].

2.2 Modular AI Capability Safety Proofs

Parametricity modularizes reasoning about the safety properties of a capability implementation. Stakeholders, assisted by theorem proving AIs [17], can mechanically prove properties of interest by instantiating the abstract monad with a suitable Dijkstra monad, which consists of a command monad (typically the environment's base IO monad) indexed by a specification monad. For example, we can prove that an implementation of the above capability signature correctly restricst access to the given directories using predicate transformers (not shown, but see [1, 10]). This indexing structure makes Dijkstra monads very flexible to a variety of reasoning techniques—they can be used for reasoning about side channel attacks, concurrency, information flow, and probabilistic programs (which could in the future be used for proofs about causal constraints).

In addition to modular proofs about capabilities, a capability signature in general could defer to the AI agent to discharge proof obligations using a dependently typed signature, e.g. one that asks the agent to provide causal proofs justifying a particular course of action before it is executed. This will require a robust world model, which can also be expressed as a collaboratively developed collection of dependently typed structures in the Bastion.

Capabilities can be composed using monad transformers [8]. If the constituent capabilities are separable, proofs will easily compose. In other cases, capabilities might interact non-conservatively.

3 Talk Logistics

The proposed talk will provide an overview of the problem space, related work, and the proposed architecture (20 minutes), then seek discussion (10 minutes) from HOPE participants who are familiar with the reasoning challenges that will come up and potential solutions from the literature that the project team should consider. We are very open to potential collaborations that might arise.

References

[1] Danel Ahman et al. "Dijkstra monads for free". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL '17. Paris, France: Association for Computing Machinery, 2017, pp. 515–529. ISBN: 9781450346603. DOI: 10.1145/3009837.3009878. URL: https://doi.org/10.1145/3009837.3009878.

[2] Cezar-Constantin Andrici et al. "Securing Verified IO Programs Against Unverified Code in F*". In: *Proceedings of ACM Principles of Programming Languages* 8.POPL (2024), pp. 2226–2259. DOI: 10.1145/3632916. URL: https://doi.org/10.1145/3632916.

[3] Ana Bove, Peter Dybjer, and Ulf Norell. "A Brief Overview of Agda – A Functional Language with Dependent Types". In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 73–78. ISBN: 978-3-642-03359-9.

[4] Patrick Cousot. "Types as abstract interpretations". In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. Paris, France: Association for Computing Machinery, 1997, pp. 316–331. ISBN: 0897918533. DOI: 10.1145/263699.263744. URL: https://doi.org/10.1145/263699.263744.

[5] Joseph Cutler et al. "Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization (Extended Version)". In: *arXiv preprint arXiv:2403.04651* (2024).

[6] David Dalrymple et al. "Towards Guaranteed Safe AI: A Framework for Ensuring Robust and Reliable AI Systems". In: *arXiv preprint arXiv:2405.06624* (2024).

[7] Henry M Levy. *Capability-based computer systems*. Digital Press, 2014.

[8] Sheng Liang, Paul Hudak, and Mark Jones. "Monad transformers and modular interpreters". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 333–343. ISBN: 0897916921. DOI: 10.1145/199448.199528. URL: https://doi.org/10.1145/199448.199528.

[9] David Lindenmayer and Ben Scheele. "Do not publish". In: *Science* 356.6340 (2017), pp. 800–801. DOI: 10.1126/science.aan1362. eprint: https://www.science.org/doi/pdf/10.1126/science.aan1362. URL: https://www.science.org/doi/abs/10.1126/science.aan1362.

[10] Kenji Maillard et al. "Dijkstra monads for all". In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341708. URL: https://doi.org/10.1145/3341708.

[11] Darya Melicher et al. "A capability-based module system for authority control". In: *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[12] Darya Melicher et al. "Bounded Abstract Effects". In: *ACM Trans. Program. Lang. Syst.* 44.1 (2022), 5:1–5:48. DOI: 10.1145/3492427. URL: https://doi.org/10.1145/3492427.

[13] Maya L. Petersen and Mark J. van der Laan. "Causal Models and Learning from Data: Integrating Causal Modeling and Statistical Estimation". In: *Epidemiology* 25.3 (May 2014), pp. 418–426. ISSN: 1044-3983. DOI: 10.1097/ede.0000000000000078. URL: http://dx.doi.org/10.1097/EDE.0000000000000078.

[14] Aseem Rastogi et al. *Programming and Proving with Indexed Effects*. Tech. rep. Microsoft Research, 2020.

[15] Peter Richards et al. "Are Brazil's deforesters avoiding detection?" In: *Conservation Letters* 10.4 (2017), pp. 470–476. DOI: 10.1111/conl.12310.

[16] Yonadav Shavit et al. "Practices for governing agentic AI systems". In: *Research Paper, OpenAI, December* (2023).

[17] Peiyang Song, Kaiyu Yang, and Anima Anandkumar. *Towards Large Language Models as Copilots for Theorem Proving in Lean*. 2024. arXiv: 2404.12534 [cs.AI].

[18] Nikhil Swamy et al. "Dependent types and multi-monadic effects in F*". In: *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2016, pp. 256–270.

[19] Nikhil Swamy et al. "Verifying higher-order programs with the Dijkstra monad". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 13. ACM, June 2013. DOI: 10.1145/2491956.2491978. URL: http://dx.doi.org/10.1145/2491956.2491978.

[20] Yong Kiam Tan, Scott Owens, and Ramana Kumar. "A verified type system for CakeML". In: *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*. IFL '15. Koblenz, Germany: Association for Computing Machinery, 2015. ISBN: 9781450342735. DOI: 10.1145/2897336.2897344. URL: https://doi.org/10.1145/2897336.2897344.

[21] Zahra Tarkhani and Anil Madhavapeddy. "Information Flow Tracking for Heterogeneous Compartmentalized Software". In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID '23. Hong Kong, China: Association for Computing Machinery, 2023, pp. 564–579. ISBN: 9798400707650. DOI: 10.1145/3607199.3607235. URL: https://doi.org/10.1145/3607199.3607235.

[22] E. Tromer and M. Krohn. "Noninterference for a Practical DIFC-Based Operating System". In: *2009 30th IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2009, pp. 61–76. DOI: 10.1109/SP.2009.23. URL: https://doi.ieeecomputersociety.org/10.1109/SP.2009.23.

[23] Robert N.M. Watson et al. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization". In: *IEEE Symposium on Security and Privacy*. 2015, pp. 20–37. DOI: 10.1109/SP.2015.9.

[24] Sven Wunder et al. *Biodiversity credits: learning lessons from other approaches to incentivize conservation*. Feb. 2024. DOI: 10.31219/osf.io/qgwfc. URL: osf.io/qgwfc.