WHITE PAPER

Formal Foundry Architecture

A Safety Layer for Generative AI

Formal Foundry Team

Formal Foundry, LLC

May 2023

Abstract

We propose a safety layer for generative artificial intelligence that combines large language models with formal verification in dependent type theory. The architecture has four components: domain formalization, generation of formal specifications, proof generation, and translation of formal statements and proofs into semi-natural language. Domain libraries encode application-specific rules; an interactive specification layer turns user requirements into proof obligations; a language model proposes proofs that an Agda type checker checks; and a deterministic translation layer makes the formal objects accessible to users and developers. Type-checker feedback supports iterative proof refinement and the decomposition of difficult goals into lemmas. The approach aims to establish correctness relative to explicit formal specifications, while the adequacy of those specifications and their domain assumptions remains a separate validation problem. We describe the components, their interactions, application-specific requirements, prototype status, and planned development. The architecture is a research and engineering proposal, with significant work remaining before broad deployment.

First published on formalfoundry.ai on 20 June 2023. Internet Archive copy (26 September 2023).

Archival edition with an added abstract, references, and marked technical notes. The original development status and roadmaps are retained.

1 Architecture overview

In the evolving realm of artificial intelligence (AI), it's crucial to ensure the safety and correctness of the systems we design [1]. This paper introduces a new architecture we've developed to do exactly that. With the marriage of large language models and dependent type theory [2, 3], our architecture aims to verify AI outputs with absolute certainty.[1]

Our architecture relies on four essential components:

- **Formalization:** This component allows us to express real-world problem statements as statements in formal language. It establishes the groundwork upon which our system operates, turning fuzzy concepts into well-defined, precise, and operable terms.
- **Theorem Generator:** Once we have our formally defined problem, the theorem generator uses an AI model to create theorems that need to be proven to ensure the correctness of a particular output. These theorems serve as targets for the proof generator.

[1] *Editorial clarification.* The guarantee is relative to the encoded specification, assumptions, and soundness of the checking environment. It does not establish that the formal model captures every relevant real-world safety requirement.

- **Proof Generator:** With the theorems in place, the proof generator steps up. It uses a language model to generate proofs for the theorems, ensuring they hold up against scrutiny. If the proof doesn't pass muster, the model gets feedback and iteratively refines the proof until it's sound.
- **Translation:** Lastly, our translation component takes over. This part translates the formal language used in theorems and proofs into human-friendly language. It allows non-experts to understand and engage with our work, bringing accessibility to the complex world of AI safety.

Throughout this paper, we'll dive into the specifics of these four components. We'll explain how they operate together to create a system that can verify the outputs of AI models with confidence.

We know our architecture is just one approach in the vast and challenging landscape of AI safety. We recognize its limitations, and we're committed to refining and expanding upon it. We invite your critique, we value your feedback, and we're excited to improve this architecture with your help.

In the pursuit of safer AI systems, our goal is to harness the immense potential of AI with precision and responsibility. We're glad to have you join us on this journey, and we look forward to working together towards a safer AI future.

2 Domain formalization

In the context of verifying the output of generative AI, domain formalization presents a unique set of challenges. Every area where we intend to implement AI systems possesses inherent complexities and adheres to a set of governing rules. For instance, if we consider the domain of accounting, the tax code serves as the rule set. Similarly, when we look at designing analog circuits, electrical engineering and the theory behind operational amplifiers serve as the guiding rule set.

A significant problem with domain formalization is defining a clear boundary between what to include as laws or rules and what to exclude. It becomes challenging to formalize a domain entirely due to the inherent ambiguity present in most fields. Furthermore, the practicality of formalizing the entire scope is often questionable, and usually, a narrower, more targeted approach proves to be more beneficial [4].

2.1 Challenges in accurate formalization

- First, it involves distilling intricate and often ambiguous domain knowledge into concrete, formal rules. For instance, consider the process of translating a complex tax code into a rule set that an AI system could understand and apply correctly. Even if a majority of the tax code is reasonably straightforward, certain aspects are subject to interpretation and contingent upon specific business considerations or legal judgments. Thus, striking a balance between adequately capturing the domain complexity and maintaining an interpretable, formalized structure becomes a challenge.
- Second, there is the problem of ensuring the correctness of these formal rules. If a particular rule is translated inaccurately or interpreted incorrectly, the ramifications could be severe. For example, an incorrectly formalized tax rule could lead to suboptimal or even harmful tax advice, potentially resulting in legal issues or financial losses for users. This issue is further complicated by the fact that generative AI systems can be exploited by those who find and understand these inaccuracies, leading to potential misuse.
- The third challenge is associated with the "expressiveness" of the formal system chosen to represent the domain rules. Traditionally, formal systems used to express logical relations were limited in their capacity to handle complex, real-world scenarios. However, with the advent of large language models, the scale of complexity has increased dramatically. These models can apply patterns from a

broad spectrum of human knowledge into a specific domain, leading to a level of creativity that is challenging to constrain within a rigid structure.

- Moreover, the creation of a formalization often requires significant domain expertise. Traditionally, these formalizations were created by domain experts, a time-consuming and labor-intensive process. While large language models can speed up this process by quickly translating rules from natural language into a formal structure, this accelerated formalization still requires a review by a domain expert, adding another layer of complexity to the problem [5].
- Lastly, the very nature of AI - its adaptability and applicability across a range of domains - means that the process of domain formalization is not a one-size-fits-all solution. The formalization for an accounting system will be drastically different from a healthcare or legal system, necessitating a custom approach for each domain.

2.2 Encoding domain knowledge

Domain formalization operates as an extensive library, encompassing definitions, proofs, and lemmas that model different safety properties of generative AI output. Having this formalization allows us to express, in concise terms, that some output of generative AI is safe in a given context. It also allows us to use a type checker to reliably verify if the argument for safety is indeed correct and self-consistent.

2.3 Variation across applications

Formalization in Agda, a dependently typed functional programming language and proof assistant [6], serves as a medium to encode all rules relevant for any given application. The formalization process is deeply contextual and will differ significantly across industries and products. Here, we explore a few examples to elucidate this:

1. **Formalization for Financial Software:** For an AI application in financial software that aids in investment or tax planning, the domain formalization in Agda would involve creating precise, mathematical models of the financial regulations, tax codes, investment principles, and statistical models that underpin investment strategies. The model might need to account for probabilities, statistical distributions, financial derivatives, and, importantly, the formalized version of financial laws and regulations. Thus, a deep understanding of financial mathematics and regulatory compliance would be required.
2. **Formalization for Healthcare AI:** In contrast, for an AI application in healthcare such as diagnostic support, the formalization process will differ significantly. The model would need to encode the medical knowledge required for the AI to function correctly, such as symptoms of diseases, medical procedures, clinical pathways, and biomedical science. The formalization might involve encoding ontologies of diseases and their symptoms, patient histories, treatment procedures, and the probabilities related to diagnoses and treatment outcomes.
3. **Formalization for Autonomous Vehicles:** In the case of an AI system for autonomous vehicles, the domain formalization might involve formalizing traffic rules, sensor data processing, vehicle physics, environmental conditions, and the decision-making process the AI uses to control the vehicle. The model would need to represent aspects such as speed limits, collision avoidance rules, right-of-way rules, as well as the effects of vehicle controls on its motion.

For each application, the domain formalization process in Agda will start with an identification of the important entities, processes, and rules within the domain. Subsequently, these will be represented in a structured way using Agda's type system and functions, ensuring that the AI's reasoning process follows these rules.

The precise shape of this formalization will depend greatly on the specific requirements of the application, the complexity of the domain, the level of uncertainty in the domain, and the potential risk and harm that can occur from incorrect AI behavior. Given the inherent complexity and differences across domains, the formalization process requires deep expertise in the specific domain, as well as experience in formal methods and type theory.

2.4 Integration with other components

For instance, during the theorem generation stage, theorems are only meaningful within the context of formalization. In the proof generation stage, the formalization must be fed to the type checker along with the attempted proof generated by the large language model to check its correctness.

2.5 Formalization roadmap

The domain formalization component is created separately for every application, and its development is based on collaboration with partner companies. Over the next year, we aim to start the formalization process as soon as we start working with the industry partners who join us in this effort. We plan to identify the scope of formalization together with these partners and work on creating the formalization ourselves. If partners wish to gain experience in creating formalizations, we welcome their contributions.

For the initial cohort, we aim to spend the beginning of the year identifying their scope, gathering all the specifications, and building each partner's formalization. The last two to three months will be spent evaluating the formalizations. It should also be noted that the specific roadmap might be different for the formalization of the different domains.

3 Theorem and formal specification generation

The primary task that the formal specification generation component seeks to address is creating a clear and specific criterion for validating the correctness of an output produced by a generative AI system. In the context of our architecture, every input given to the AI system should ideally correspond to a unique formal specification, or in simpler terms, a clearly articulated 'rule' that can be used to judge whether the output provided by the AI is correct or not.

This 'rule' or formal specification, which is essentially a theorem, has to be generated each time a new input is given to the AI system. The theorem serves as a benchmark to evaluate the AI's output, similar to having a checklist or a set of guidelines to assess a piece of work. In the absence of this theorem, it becomes difficult to ascertain whether the AI's output is correct or safe.

In a nutshell, the task of the formal specification generation component is to ensure that for each unique input to the AI system, a corresponding unique formal specification or theorem is created. This theorem lays the groundwork for validating and verifying the safety and correctness of the AI's output. This process is crucial to the effective operation of our overall architecture, ensuring its robustness and reliability across different applications and industries.

3.1 Challenges in generating correctness specifications

Creating and managing formal specifications of output correctness presents several challenges:

1. **Variability in Input Style:** Large language models typically accept inputs in natural language, which can be less precise than formal problem descriptions. This variability can make it difficult to generate exact formal specifications, particularly when user inputs are varied and less structured.

2. **Interaction Complexity:** For systems that rely on user input, obtaining precise and formal descriptions requires engaging the user in complex interactions. These interactions need to guide the user towards providing precise details and clarifying ambiguities, which can be an intricate and time-consuming process.
3. **Formalizing Ambiguities:** Natural language is often nuanced and full of ambiguities. Converting these nuanced inputs into strict, formal specifications that can be handled by proof assistants is a non-trivial task, often requiring advanced language processing techniques and heuristics.
4. **Requirement of Expertise:** Generating formal specifications usually necessitates an understanding of formal methods or programming language semantics. The majority of users may not have this expertise, which adds another layer of complexity to the problem.
5. **Domain Specific Challenges:** Each domain or industry will have its unique requirements, further complicating the process of generating universally applicable formal specifications. The component must be flexible enough to accommodate these unique needs, which requires significant foresight in design and implementation.

Creating formal specifications of output correctness is a simpler task for systems where the AI's input is generated automatically, or the specification layer is hardcoded in a deterministic manner. However, the challenge becomes complex for systems where users provide inputs in natural language format.

3.2 Interactive specification inference

The formal specification generation component establishes a structured dialogue between the user and the AI to generate formal specifications, which outline the 'correctness' of AI-produced output. Notably, this process is designed to be user-friendly, requiring no prior knowledge of formal languages like Agda from the user's side.

This component consists of a user interface (front-end) and a logic layer (back-end). The user interface presents the users with questions that delve into their intended output, which the large language model has generated. As users respond, their answers are fed into the back-end logic layer.

In the logic layer, the user's answers shape the formulation of subsequent questions. This iterative process refines the understanding of the user's requirements and the expected AI system output with each cycle. Concurrently, the logic layer constructs a formal specification in the form of a proof assistant statement.

An integrated type checker within the component ensures the formal specification's well-formedness and accuracy at each iteration.[2] Thus, through the continuous user engagement and refinement of the understanding of the task, this component allows for the generation of precise formal specifications that accurately reflect user intent, all without requiring the user to understand formal languages.

3.3 Variation across applications

The implementation of the formal specification generation component can significantly vary from one application or industry to another, depending primarily on how the inputs for the AI are generated - either directly from the user in a natural language form, or prepared automatically.

In scenarios where the user provides inputs in natural language, the component engages in an iterative, interactive process with the user. This engagement involves the AI posing additional questions to the user to help extract precise requirements and specifications for the desired output. The extent of

[2] *Editorial clarification.* Type checking establishes well-formedness, not fidelity to user intent. That fidelity requires validation of the specification. The preceding use of "unique" means an input-specific specification, not mathematical uniqueness.

this iterative interaction and question-asking process might vary. For instance, in domains where the initial task description is comprehensive, the need for further questions could be limited to specific clarifications. Conversely, in cases where the initial input lacks explicit detail, the component might rely heavily on the iterative question-asking process to generate the formal specification.

For applications where the AI's input is prepared automatically, the formal specification generation process can be considerably more streamlined and deterministic. Given the structured nature of such inputs, the translation into formal specification can be accomplished through a deterministic algorithm.

These varying implementations require the development of a flexible toolkit that allows customization of the component as per the specific needs of each application. This toolkit will comprise common tools for specification generation, which can be customized and scaled to suit specific applications.

3.4 Use of specifications across the system

It uses domain formalization to reduce the size of the theorem expressing the desired output correctness. The specifications generated by this component serve as input for the proof generation layer.

Additionally, the deterministic translation component can convert the formal specification into a human-readable version. In certain applications, allowing the user to review the formal specification in this format can be beneficial to ensure the accurate extraction of their intention.

3.5 Development status and roadmap

The formal specification generation component is currently in the experimental stage. We have already conducted preliminary tests and observed promising results. Notably, we have seen success in obtaining formal specifications for web application state management. The language model was adept at guiding users through the intricate process of specifying behavior from a high-level perspective and then gradually addressing each detail and ambiguity.

Our plan for further development, testing, and refinement of the formal specification generation component follows a structured timeline and includes direct collaboration with our industry partners. This comprehensive roadmap is structured as follows:

Month 1-4: During the first four months, our main objective will be the development of a comprehensive toolkit that serves as the foundation for the formal specification generation layer. This toolkit will be designed to be versatile enough to facilitate the creation of formal specifications across various applications.

Month 5-6: Post toolkit development, we will enter a phase of extensive internal testing and calibration. The objective here is to identify and rectify potential bugs and streamline the processes. We will conduct tests across a range of applications to ensure the toolkit's robustness and versatility.

Month 7-9: This phase will focus on training our AI models to use this toolkit effectively and generate well-structured, relevant formal specifications. As the AI model interacts with the toolkit, we expect that it will improve its understanding and proficiency in formal specification generation.

Month 10-12: The final quarter of the year will involve extensive collaboration with our partners. By this time, we expect to have a functional model that can generate formal specifications efficiently. Our partners will then test this model within their specific applications and provide us with feedback. Working directly with our partners will provide us invaluable insights into how the formal specification generation component operates in diverse real-world scenarios. We anticipate that this feedback will provide us with a nuanced understanding of the strengths and areas of improvement of our component. The final month will be dedicated to incorporating this feedback and making the necessary adjustments to the component.

4 Proof generation

The proof generation component is a crucial element for the validation of generative AI's output. Given a theorem or assertion of safety, the challenge lies in producing a concrete proof that confirms the correctness of this assertion. To clarify, if a generative AI produces an output, we need to validate that this output is indeed safe, as per the theorem or assertion provided by the theorem generation component. The proof generation component addresses this fundamental need.

4.1 Challenges in proof generation

The primary difficulty in addressing the problem of proof generation in generative AI arises from the inherent heuristics in the nature of such AI models. These models can produce outputs that mimic safety convincingly, presenting significant challenges for verification mechanisms. There are several reasons why this problem is hard to tackle:

1. **Deceptive Correctness:** The outputs from generative AI can appear correct and safe, even when they are not. This "mimicry" can exploit blind spots in the verification process, creating outputs that seem safe on the surface but are potentially harmful.
2. **Binary Nature of Correctness:** Correctness is not a gradual or relative concept - a solution is either correct or it isn't. This binary nature means there's no room for approximation in our verification process. Ensuring absolute correctness therefore requires a completely reliable, objective mechanism that leaves no room for interpretation.
3. **Layering Heuristics:** Attempting to add another heuristic layer on top of generative AI to ensure safety might not suffice. This approach could potentially reduce the degree of incorrectness, but it might also create a false sense of security, treating an unsafe system as safe. It's essential to remember that safety is non-negotiable, and even a small area of unsafety can lead to significant problems.
4. **Gradual Improvement Difficulty:** When an incorrect proof is detected, it can be challenging to iteratively improve the proof until it becomes correct. The model has to interpret feedback from the type checker and refine its output, which can be an intricate and complex process.
5. **Complexity of Real-World Scenarios:** Proofs in real-world applications can be incredibly complex, requiring the generation of intermediate theorems (lemmas) to break down larger problems into smaller ones. Implementing an automated process to handle this effectively is challenging.

Therefore, the development of a reliable proof generation component in AI safety is both challenging and crucial to ensure that generative AI systems operate safely and correctly.

4.2 Component design

The proof generation component is an integral piece of our architecture that is designed to verify the safety and correctness of AI outputs. Its key purpose is to take a statement generated by the AI system and produce a concrete proof that either validates or refutes that statement's correctness.

This component is the amalgamation of a large language model (LLM) and a type checker, orchestrated to work in an interactive feedback loop. The process is initiated when the AI system generates an assertion of safety or a theorem pertaining to its output. This statement is then ingested by the proof generation component.

At the core of this component's operations, the LLM is first tasked with generating an initial version of the proof. The produced proof is subsequently assessed by the type checker, a module dedicated to

validating the proof's correctness [7, 8]. If the proof aligns with the correctness criteria, the process halts, marking the proof as valid. If not, the type checker provides feedback to the LLM, which then refines the proof. This feedback loop continues until a correct proof emerges, ensuring an iterative, responsive mechanism for proof generation.

A critical aspect of this component's architecture lies in its capacity to prove incorrectness. This ability comes into play when the system fails to produce a correct proof. In such cases, the LLM can generate a counter-theorem, essentially attempting to prove the opposite of the original statement. This capability provides explicit feedback on the specific areas of the initial output that failed, which guides the AI system to generate an improved output.[3]

The proof generation component's design incorporates an ability to generate lemmas, or auxiliary theorems, that can break down complex problems into smaller, more tractable ones. This hierarchical approach to problem-solving is a key strategy to circumvent the inherent limitation of LLMs in handling vast contextual spaces. By providing a method to split larger problems into smaller ones, the system can manage the proof generation process more effectively, given the limited context available to the LLM at any one time [9]. This ability to generate proofs for smaller components and then combine them into a proof for the larger problem greatly enhances the efficiency of the system.

4.3 Variation across applications

Despite the wide array of potential applications for generative AI, the proof generation component maintains a high degree of universality, with little to no need for significant customization across different deployments. This universality is largely attributable to the use of Agda, a dependently typed functional programming language that we leverage for proof generation.

Agda's expressive power stems from its type system, which enables us to formulate and prove theorems within the language itself, providing a consistent framework for all applications. Agda's types are predicated on intuitionistic logic, making them ideal for our task of proof generation. They allow us to constructively establish the truth or falsity of a theorem, which aligns with our goal of ensuring the safety of generative AI outputs.

So, whether the generative AI is being used in legal tech, pharmaceuticals, finance, or any other industry, the proof generation component remains largely constant. There may be minor adjustments to configurations or parameters based on specific theorems or domains that are more prevalent in a given application. However, these are primarily tuning nuances rather than fundamental changes to the component.

4.4 Interactions with other components

The proof generation component is crucial at multiple stages of the workflow. It interacts with the theorem generation component to validate the correctness of the theorems produced. After generating a proof or refutation, it can provide feedback to the initial AI model to refine the output. The proof generation component can also be utilized at the specification stage, ensuring that the specification is not self-inconsistent even before the generative model begins its work.[4]

[3] *Editorial clarification.* Proof search is not guaranteed to succeed or terminate. Failure to prove a proposition is not evidence of its negation: a refutation needs a separately checked proof. Intuitionistic type theory does not make every proposition decidable.

[4] *Editorial clarification.* Checking a statement or an individual proof is not a general consistency check for the specification or its axioms. Such claims require separate justification; unchecked assumptions and incomplete proof terms must not be treated as established evidence.

In the broader architectural view, the proof generation component might be employed at various points of the workflow, serving different purposes, reinforcing the overall safety of the generative AI system.

4.5 Development status and roadmap

The current version of the proof generation component is already generating proofs and is the centerpiece of our main demos. We have developed and are continuously expanding a database of Agda problems to test the component and assess the ability of large language models to solve them.

While the current performance may not be suited for business-relevant applications, there is a clear roadmap for the component's development. We plan to expand our problem database, add new prompts and examples, and implement a more complicated workflow to interact with large language models.

One of the first tasks in the upcoming months will be to automate the generation of lemmas, a feature that has shown promise in initial experiments. With this improvement, the performance of the proof generation component is expected to increase significantly, benefitting all our partners.

As proof generation using large language models is an active area of research [10, 11], we will closely monitor advancements in the field and incorporate new techniques and ideas into our tool.

5 Translation to semi-natural language

5.1 The comprehension barrier

The cornerstone problem that the semi-natural translation of Agda theorems and proofs seeks to address is the comprehension barrier for the average developer or user. Dependently typed languages like Agda are not commonly understood in the industry, despite their extensive usage in representing both formal specifications and their proofs. The lack of widespread knowledge about such languages poses a major problem when users or developers need to understand the arguments and specifications handled by systems built with Agda.

Consider a system that provides tax advice using generative AI. There may come a time when a practice once considered legal gets flagged as possibly illegal. Understanding why this change occurred can be pivotal for both users and developers. However, the explanation could be buried in hundreds of lines of Agda code, which can't be deciphered without expert knowledge in Agda. Semi-natural language translation can provide valuable insight, helping stakeholders understand what's happening.

Not addressing this issue could result in a lack of transparency and understanding, turning the system into a "black box" for users. This can hinder the larger goals of AI safety and assurance. As AI continues to evolve and grow in complexity, it's important to clearly distinguish when a system operates based on creative, but potentially unsafe heuristics, versus when it operates in a predictable and deterministic manner.

The semi-natural translation of formal specifications and proofs is crucial for making this distinction. Additionally, it could also pave the way for companies to develop in-house expertise in dependent type languages, simplifying the comprehension of formal proofs and specifications, and thereby contributing to better understanding and safer utilization of AI systems.

5.2 Translation challenges

The primary challenges in translating Agda theorems and proofs into a semi-natural language lie in the intricacies of both the source and target languages. Despite Agda having clear semantics for the

majority of its constructs, generating semi-natural representations might require defining arbitrary rules for specific constructs. Natural languages are often characterized by heuristic-based generation methods that, while successful, are non-deterministic and may introduce inconsistencies - a risk we cannot afford in safety components.

Moreover, the diversity of formal systems, such as differences among Agda, Coq and HOL, can pose challenges to translation efforts. Our initial focus will be on Agda, but with certain modifications, the techniques should be translatable for languages with similar type systems.

5.3 Component design

The semi-natural translation component is a critical toolset within our system that interfaces directly with the Agda type checker, facilitating a bridge between formal logic and human language.

At the heart of the toolset, we have the core translation component, designed to ingest Agda theorems and proofs and translate them into semi-natural language. Upon receiving Agda code, the component interacts with the type checker to parse the Abstract Syntax Tree (AST) of the input. Each node of the AST is then evaluated based on a series of hardcoded rules, which serve as a translation guide for Agda constructs [12]. For instance, Sigma and Pi types in Agda, which carry clear semantics as existential and universal quantifiers, would be translated according to pre-set rules.

However, it's not a one-size-fits-all situation. Given the diverse nature of logical domains, custom rules will be needed for domain-specific constructs that are introduced during the formalization stage. Such domain-specific rules provide an opportunity to incorporate specific terminologies or jargon from the field, thereby enhancing the understandability of the output for professionals in that field.

Moving beyond translation, the output is then structured into a format that is easy to parse and navigate, such as a JSON file. This structure would entail a layer of semantic organization, enabling a more intuitive exploration of the translated content.

To make the semi-natural language representation accessible and user-friendly, we plan to implement a frontend widget. This widget, which can be incorporated into the interface of systems using our safety layer, receives the structured output and displays it interactively for users or developers. The widget would likely be developed using popular frontend technologies such as React or Elm.

5.4 Variation across applications

In different applications or industries, the semi-natural translation component may be adapted or modified in a few key ways. First, annotations within the formalization can be added to introduce domain-specific lingo and vocabulary. This enables the component to generate translations that use terminology familiar to the users in a specific industry, which can significantly improve the comprehensibility of the translations.

Second, the frontend component can be customized to fit with the aesthetic and usability needs of different systems. Depending on the needs of the users and the complexity of the specifications being translated, the frontend could be deployed in a simpler form or a more in-depth version.

Third, the treatment of generic Agda constructs can be configured to best fit the domain of the application. For instance, how logical quantifiers, logical arguments, and counterexamples are represented can be adjusted to best suit the needs of a particular use case. This allows for the development of a diverse set of strategies to tackle domain-specific translation challenges.

The level of customization required for this component in different applications will largely depend on the complexity of the specifications given to the generative AI and the familiarity of the users with the domain. For instance, if the specifications are typically constructed with well-defined building

blocks and do not involve complex logical predicates, it may be unnecessary for the users to inspect the full specification. However, if the specifications can be more complex, involving user-defined logical relations or properties that are not easily enumerable, then the ability to understand the underlying formal representation might be critical.

The preferences of different users may also influence the workings of this component. While average users may not often require detailed insight into the specifications, expert users might find this insight invaluable, especially if they are using the product to produce highly tailored and complex output. Consequently, the frontend needs to be designed in a way that it can provide useful information to both average and expert users, without overwhelming the former or under-serving the latter.

Even if the semi-natural translation layer is not directly exposed to users, it can still be a useful tool for developers during the development and maintenance phases of a system. It can help to ensure that updated knowledge remains consistent with the existing parts of the system and can aid in debugging. Thus, this component can be valuable both within and outside the context of a product.

5.5 Development status and roadmap

Current development status

At present, we have a basic proof of concept internally developed in Agda. This tool recognizes some essential Agda constructs, translating them into a semi-natural language. One can view examples of this generated language representation in our repository, where we've applied it to a formalization of the Elm language. It's worth noting, these representations effectively translate statements about formalized syntax into sentences about web applications written in Elm.

Key milestones

The significant achievement thus far has been the successful implementation of our proof of concept, which provides us with a foundational structure to build upon. The proof of concept, being used internally within Agda, has demonstrated effective performance with the Elm language formalization.

Roadmap

Short-term (1 - 3 months): The immediate plan is to re-implement the semi-natural language generation as a Haskell process that interacts directly with the Agda compiler. This refactor will enable us to process Agda abstract syntax tree (AST) more effectively.

Mid-term (3 - 6 months): We aim to tailor the translation layer to better fit the needs of our partners. During this period, we will be focusing on assessing different strategies for generating such representations across the various industries.

Long-term (6 - 12 months): The overall goal is to continue refining and expanding our translation rules to cover a broader spectrum of Agda constructs.

Long-term evolution

In the long run, we foresee the semi-natural translation component's capabilities growing exponentially. As more reliable methods of generating natural language become available, we expect that our system will produce increasingly comprehensible representations of proofs and specifications. Moreover, the transfer of techniques developed in one industry to another is likely to enrich the toolkit for implementing safety architectures based on formal methods. This evolution, however, largely hinges on the advancements in safe natural language generation methodologies and the increased acceptance of formal methods in various industries.

References

[1] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané. *Concrete Problems in AI Safety*. 2016. arXiv: 1606.06565 [cs.AI].

[2] P. Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Notes by Giovanni Sambin. Naples: Bibliopolis, 1984. URL: https://archive-pml.github.io/martin-lof/pdfs/Bibliopolis-Book-1984.pdf.

[3] P. Wadler. "Propositions as Types". In: *Communications of the ACM* 58.12 (2015), pp. 75–84. DOI: 10.1145/2699407.

[4] J. M. Wing. "A Specifier's Introduction to Formal Methods". In: *Computer* 23.9 (1990), pp. 8–22. DOI: 10.1109/2.58215.

[5] Y. Wu, A. Q. Jiang, W. Li, M. N. Rabe, C. Staats, M. Jamnik, and C. Szegedy. "Autoformalization with Large Language Models". In: *Advances in Neural Information Processing Systems*. Vol. 35. 2022, pp. 32353–32368. DOI: 10.52202/068431-2344.

[6] U. Norell. "Towards a Practical Programming Language Based on Dependent Type Theory". PhD thesis. Chalmers University of Technology and Göteborg University, 2007. URL: https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf.

[7] G. C. Necula. "Proof-Carrying Code". In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1997, pp. 106–119. DOI: 10.1145/263699.263712.

[8] S. Polu and I. Sutskever. *Generative Language Modeling for Automated Theorem Proving*. 2020. arXiv: 2009.03393 [cs.LG].

[9] G. Lample, M.-A. Lachaux, T. Lavril, X. Martinet, A. Hayat, G. Ebner, A. Rodriguez, and T. Lacroix. "HyperTree Proof Search for Neural Theorem Proving". In: *Advances in Neural Information Processing Systems*. Vol. 35. 2022, pp. 26337–26349. DOI: 10.52202/068431-1910.

[10] J. M. Han, J. Rute, Y. Wu, E. W. Ayers, and S. Polu. "Proof Artifact Co-Training for Theorem Proving with Language Models". In: *International Conference on Learning Representations*. 2022. URL: https://openreview.net/forum?id=rpxJc9j04U.

[11] S. Polu, J. M. Han, K. Zheng, M. Baksys, I. Babuschkin, and I. Sutskever. *Formal Mathematics Statement Curriculum Learning*. Preprint, 3 February 2022. 2022. arXiv: 2202.01344 [cs.LG].

[12] A. Ranta. "Grammatical Framework". In: *Journal of Functional Programming* 14.2 (2004), pp. 145–189. DOI: 10.1017/S0956796803004738.