Generic Models and The Hiconic Platform

The Invention

Our development revolves around novel concepts for models in the realm of programming languages, currently Java and TypeScript. By extending existing type systems with normalized data types for property containers, and incorporating robust reflection capabilities alongside homoiconicity, our technology effectively bridges the gap between domain-agnostic and specific domain requirements. With this technology at its core, we have successfully developed a modular web platform that is capable of accommodating a diverse array of applications.

For the largest part of the development the technology and the web platform was proprietary software. Finally in 2023, it went fully open source under the LGPLv3 license and is maintained on Github.

Drawing on the concept of homoiconicity, we are confident that our technology can also enhance interactions with Large Language Model (LLM) based AIs. This becomes especially pertinent in situations where interactions with an AI necessitate exacting precision and produce actionable outcomes, or in instances where the AI is required to engage with or manipulate traditional IT systems.

Practical Relevance

To underscore the practical potential of our technology in addressing real-world challenges, we would like to contextualize it within the framework of contemporary programming languages and the tangible requirements of today's IT landscape.

In-Process Function Calls

Modern programming languages have been developed to facilitate the handling and transfer of structured data. They achieve this by utilizing functions within a process that operates on a single computer. This design reflects a focus on streamlining the interaction with and manipulation of data in a computing environment that is typically confined to an individual machine.

In these programming languages, a traditional function call mechanism is used, grounded in the utilization of an in-memory call stack. This stack is essential for transmitting arguments, setting a return address for post-execution continuation at the call site, and managing the function's results. To call such a function a caller has to provide a name and positional arguments in the form of a list. The name resolves to an address and the arguments will be orderly pushed to the stack as expected by the function.

This call mechanism, while optimizing for execution speed due to its simplicity, does come with certain compromises. For instance, the use of named arguments could offer greater resilience, but this is not typically leveraged in the standard approach. Additionally, the process of passing arguments is somewhat opaque to the programmer, making it less adaptable for intercepting and managing cross-cutting aspects.

Remote Function Calls

The classical principle of function calls turns into a tangible disadvantage in contemporary scenarios where program execution is not confined solely to in-process functions. Much of the functionality now resides remotely, either in different processes on the same machine or across various machines in a network. This shift in execution dynamics challenges the traditional function call mechanism, which is primarily designed for a more localized, in-process environment.

Programmers face distinctly different challenges based on whether they are dealing with in-process functions or remote functionalities. Remote function calls introduce a range of cross-cutting concerns absent in in-process calls. These encompass managing network connections, handling authentication and authorization, implementing retry logic, and addressing serialization, encoding, escaping, transport, error handling, asynchronicity, cancellability and protocol management. Each factor significantly complicates remote function calls, setting them apart from in-process counterparts. Programmers often tackle these cross-cutting aspects with varying degrees of focus, sometimes even overlooking them. Frequently, these aspects can obscure the core implementation of the functionality, whether it involves making or receiving remote calls. The undesired overall effect is an increase of software entropy with all its negative consequences for understanding, safety, stability, portability and maintainability.

Homoiconicity as Solution for Remote Function Calls

This situation further exposes the limitations inherent in the classical function call paradigm, particularly evident in the divergence between how functions and data are represented. In modern programming languages, data is often structured through type-safe polymorphisms, characterized by attributes (also known as fields or properties). In contrast, function calls do not follow this pattern. This disparity in representation, or denormalization, significantly influences the design of remote functions. For instance, in the realm of web APIs, we observe that functions are addressed differently compared to the data they handle. URLs are commonly used to identify functions, while URL parameters and HTTP message bodies are

employed to pass arguments. This practice reflects the continued separation in the treatment of functions and data, a vestige of traditional programming methodologies.
To overcome the denormalization between data and function representations, we propose a unified approach using standardized entity data types within models. These types encompass a qualified name, a set of super types, and publicly accessible properties. The properties can be of primitive types (like boolean, integer, double, string, date), entity types, or collections (like list, set, map) of these types.

The simplicity of these entity types facilitates efficient reflection, allowing introspection into the structure of an instance or type. This capability enables the enumeration of super types and properties, as well as generic read/write operations on instance properties. The read and write access on properties is generically interceptable to implement further features like manipulation tracking for concepts like MVC and others.

This design allows generic algorithms to handle information domain-agnostically, addressing the previously discussed cross-cutting aspects. Data is already often designed in this manner and applying the same principles to represent functions offers several benefits.

In this approach, each function just like data is represented as an entity type. The entity-type's qualified name identifies the function, while its properties act as named arguments. A response type is associated with such a type in order to describe what the according function will return. Abstract supertypes can be used to bundle functions on different levels offering much more flexibility than bundling methods with a traditional class. This flexibility also takes account of function overrides. Additionally, handling default values becomes more flexible with named arguments than with positional ones. This normalization of data and code can be also understood in terms of homoiconicity or as a higher level of goedelization.

Given that an entity type can encompass numerous properties, it inherently supports a corresponding number of arguments. In this framework, the implementation linked to the entity type is thoroughly standardized, as it consistently receives only a single actual parameter: the instance of the function call, complete with properties representing its arguments. It also uniformly returns a singular value. Consequently, this approach results in a standardized format for pointcuts and interceptors, which are solely reliant on polymorphic types. Such a structure notably diminishes the ambiguity often associated with aspect-oriented programming in the context of classical function calls.

Under this concept the entire instance can be handled in a normalized way to simplify cross-cutting as required for the remote handling. For remote functionalities, serialization processes can uniformly handle the entire structure, eliminating the need to distinguish between parts addressing the function and those representing arguments. This efficiency eliminates the necessity for multiple URL paths, requiring only a single protocol endpoint.

A Homoiconic Application Platform

Utilizing the foundational type-system and its reflective capabilities, our platform offers a diverse range of domain-agnostic algorithms and APIs. These enhancements span a variety of functionalities, including the resolution of cascading metadata across models, types and properties, advanced cloning/mixing capabilities, data trimming and incremental in-memory persistence with comprehensive query support to name the most important. These features are fully integrated and operational within our web platform, which adopts a holistic modeling approach. This approach covers everything from configuration and reflection to persistence and service domains. Furthermore, the platform is designed to automatically interface service domain models with established standards like Open API and GraphQL, ensuring broad compatibility and ease of integration. We also integrated our modeled services into command line tools used for CICD pipelines, building and platform setup.

The Future

We are keen to share these innovations and successes with a wider audience and are looking to expand and evolve our work in several potential key areas:

1. **Continued development and strengthening of our web platform**
   We aim to further refine and enhance the capabilities of our existing web platform and with it promote its principles.
2. **Creating a minimal function host for fast bootstrapping of modeled microservices**
   As an alternative for our modular platform monolith we want to support a microservice carrier framework. Being very lightweight and less reflected, it should just host one or a few modeled services per scaled instance and connect it with endpoints for either message queues or forms of web RPC.
3. **Research in which ways AI can contribute or profit from the principles**
   It turned out that ChatGPT was very easily instructable to build entity types from a few describing words. That indicates that the normalized structure of our entities could be very helpful to leverage AI co-piloting.
4. **Adaptation of our type-system for additional programming languages**
   We plan to develop a core version of our type-system that can be integrated with other suitable programming languages.
5. **Invention of a type-safe and scalable exchange format for complex graph data**
   Building on our type-system we intend to design a new format optimized for messaging, RPC, event systems, and configurations. This format would be a robust alternative to current standards like XML, JSON, and YAML adding advanced features such as forward references, value pools to support compact

entity representation and compare stabilization through referencing instead of nesting.

6. **Extension of an existing programming language**
   We aim to enhance a current programming language to natively support our type-system and reflective capabilities at both the compiler and runtime levels.
7. **Development of a new programming language based on models and homoiconicity**
   We envision creating a programming language that inherently incorporates the principles of models and homoiconicity. This language would focus on a meta-model and an instruction-model as the primary layers of declaration and execution, subsequently integrating user-friendly grammars that parse into these models.