Invention of Generic Models

Our invention fundamentally consists of a reflective type system for model entities, which we initially developed as an extension for Java and TypeScript. The invention was a logical progression, considering the needs of modern software systems and leveraging experiences from several decades. The result is a consolidation, formalization, and refinement of typical practices that the community has been using in an immature form. A comparison can be drawn to the relationship between pseudo-object-oriented programming in C with structs and the invention and use of a dedicated object-oriented language like C++. A similar example is the transition from laboriously constructed SmartPointer solutions in C++ to the natural references in Java and similar languages. The dedicated invention sets essential guidelines for the paradigm from the outset, turning cumbersome extras into elegant necessities.

The invention was initially developed as proprietary property. In 2023, it was fully open-sourced under the LGPLv3 license and is maintained on GitHub (https://github.com/orgs/hiconic-os). The open-source code is extensive, ranging from basic language extensions, tooling, and dependency management to an application platform and its standard modules. What was achieved under proprietary conditions is remarkable, but it must also be honestly admitted that commercial conditions contributed to the preparation for a community (especially documentation) being outdated.

With the departure from the proprietary context, the question now arises as to how the project can continue to be driven and supported. In one case there could be an integration into a runtime system like quarkus or spring-boot. In another case, a native integration into a modern programming language could take place to unleash its full potential.

Introduction to the Novel Type System for Models

The type system with multiple inheritance is based on interface definitions, which describe entities with typed properties. In Java, properties are represented by a pair of getter and setter due to the lack of a corresponding language feature. The properties each have a type with the following possibilities:

- *Elementary Types*
  - *Primitive Types*
    - **boolean**
    - **integer**
    - **long**
    - **float**
    - **double**
    - **decimal**
    - **date**
  - *Custom Types*
    - **entity**
    - **enum**
- *Generic Collection Types for Elementary Types*
  - **set**
  - **list**
  - **map**

The type system generally aims to move beyond the encapsulation of object-oriented programming and instead promote the separation of data and code. There are few exceptions where default methods on entity types use reduced encapsulation for reasons of performance and elegance.

The normalization of data and code in terms of homoiconicity aims to overcome this separation of data and code at a higher level and move towards functional programming for models.

Compiler for Entity Interfaces

The use of interfaces for the definition of entities leads to automatic implementation by a compiler. While interfaces expand the developer's design space through multiple inheritance, automatic implementation simultaneously takes on redundant work. Further substantial advantages and performances arise from the automatic implementation. This includes detailed and highly efficient reflection. Each user-defined type automatically receives a reflective instance of EntityType, accessible as the T literal in the interface's namespace. The T literal is comparable to the class literal of Java Reflection. This instance can explore inheritance, list properties, and create instances. The reflection of properties allows learning the type of the property and enables generic read and write access to concrete property values on associated entities. Unlike known Java Reflection, the generic access through compiled reflection for entity types is more elegant and has no speed losses compared to direct explicit access via setter or getter. Based on this generality, efficient domain-unspecific algorithms can be written, as urgently needed for current IT needs. A tangible example is serialization into generic structural formats like JSON or YAML. Another advantage of the automatic implementation of entity types is the support of Property Access Interceptors. These can generally be inserted into the read and write access to properties. In this way, cross-cutting concerns such as Lazy Loading and Manipulation Tracking are represented. A special achievement of automatic implementation is that the property fields are also capable of holding alternative evaluable entities instead of the actual type-safe values. In this case, a Property Access Interceptor can perform dynamic evaluation with the result of the actual value, thus supplying the type-safe getter.

In the case of the solution for Java, the compiler works at runtime based on Java bytecode generation. In the case of TypeScript, we used the GWT Transpiler to reuse our core implementations of the Java solution to initially obtain a pure JavaScript library. Here, GWT code generators are used at compile time to implement the entity interfaces. However, JavaScript codes are also generated to create the interfaces and their implementation at runtime. The GWT Transpiler standardly produces compact, obfuscated JavaScript, except for namespaces of public APIs. To elevate the result of the GWT Transpiler for our public APIs and entity types to the level of type safety, we wrote another generator for generating associated TypeScript declarations.

Models for All Purposes and Homoiconicity

The models modeled with the entity types of the type system are expressly intended for all purposes, thus generalizing many existing model systems. The latter are unfortunately one-sidedly tied to specific domains such as serialization, RPC, Web API, and persistence, often bringing translation efforts and limited performances. Nevertheless, our generalized approach serves the isolated use cases of existing model systems. In the sense of a holistic system, domains that should promote the homoiconicity of the system are served with special care. These include reflective, configurative, and functionally evaluable models with connected expert systems, as the following examples show:

Model	Categorization	Description

Model	Categorization	Description
meta-model	reflective	Can describe all models, including itself, and serves as input for compilers and transpilers as well as input and output for model transformers for UML, XSD, Open Api Schema, GraphQL Schema
meta-data-model	configurative	Base model for metadata as it can be attached to elements of the meta-model for description, persistence connection, expert binding, introduction of cross-cutting concerns, etc.
value-descriptor-model	functionally evaluable	Base model for evaluable types that can be used as dynamic substitute values in entity properties.
manipulation-model	functionally evaluable	Describes instantiations and changes of individual entities to make these processes reproducible for persistence, undo/redo, and MVC, thus also supporting event-source databases.
deployment-model	reflective	configurative
service-api-model	functionally evaluable	Base model for Service Requests, which can be used in APIs for CLI, Web, or Queues.
query-model	functionally evaluable	Model for querying persistence in OO or SQL databases.

The consistent use of the new type system for fundamental models to promote homoiconicity clearly highlights its special significance for the normalization of software systems. This normalization enables significant reduction in redundancy and software entropy. The interplay of the homoiconic models and their expert systems with each other and the possibility of applying them to themselves is an effective basis for reflective, (self-)instrumentable, portable, and distributed systems. The many prepared generic algorithms based on reflection also contribute to keeping the system logically abstract and portable.

The Pivotal Positioning of the Models

The paradigm that emerged from the type system applies a "models first" strategy. So instead using a grammar as origin of formal declaration of a domain we use a model. The models are thus pivotal. Comfortable and expressive grammars can be added as needed afterward. A parser for such an additional grammar always generates instances of the associated model. These can then be further processed with the appropriate experts. In this way, there can also be different grammars with different focuses for the same model.

Use Case Object-Oriented In-Memory Database

Since reflected entities are ideal for generic searches and changes, an in-memory OO database was developed based on the type system, the query-model, and the manipulation-model. Together with the associated session-api, this persistence can be accessed in the same way as an alternative SQL database.

Use Case Distributed Modular Application Platform

With the help of the homoiconic approach, an application platform for distributed systems was developed. The platform manages and reflects models and configurable system components from modules in its Cortex Persistence. Essential basic models of Cortex Persistence include the meta-model, the meta-data-model, the deployment-model, and the service-api-model. Through Cortex Persistence, modeled functional system components and models can be networked and activated. The resulting system and application functions are logically callable via extensions of the service-api-model. Various automatic endpoints for Web/HTTP, Message Queues, and CLIs enable the call by remote components (other processes and machines).

The application platform can be extended by custom models and modules using dependency management. This allows the platform to host and provide any generic or domain-specific functionality in a network. Anyone can develop modules and models for this extension. The platform thus leaves behind the typical silo effect.