An Ecosystem Architecture for the Magnolia Programming Language

Benjamin Chetioui[1][0000−0002−7458−9079], Mikhail Barash[1][0000−0002−7067−2588], and Magne Haveraaen[1]

University of Bergen, Norway
{benjamin.chetioui,mikhail.barash,magne.haveraaen}@uib.no

Abstract. We describe our envisioned architecture for an ecosystem built around the Magnolia research programming language. The compiler for the language is built by interconnecting a core monolithic chunk with modular, extensible program transformations. "Plugins" are then constructed around a common input format, including composable program transformations—both at the syntactic and semantic level. This submission is a poster submission.

Keywords: Magnolia · Tooling Ecosystem.

1 Introduction

Magnolia [1] is designed as an embodiment of a language for generic programming. It offers no primitive types (beyond predicates), and is meant to be parameterized by a backend programming language, and data structures implemented in that language. Magnolia allows expressing generic algorithms in their most general form, and specifying their syntactic and semantic requirements explicitly. The language is based on the theory of institutions [3]. Magnolia code is written in different kinds of modules that mix purely abstract specifications of types and operations with their concrete implementations. Listing 1.1 shows the specification of a `Semigroup` in Magnolia.

Listing 1.1. Specifications in Magnolia.

```
signature Magma = {
  type T;
  function bop(t1: T, t2: T): T;
}

concept Semigroup = {
  use Magma;
  axiom bopIsAssociative(t1: T, t2: T, t3: T) {
    assert bop(t1, bop(t2, t3)) == bop(bop(t1, t2), t3);
  }
}
```

Concepts [4] constitute the core building blocks in Magnolia programs. A concept consists of abstract types and operations, along with axioms defining

semantic requirements on them. Listing 1.2 shows how to use these axioms to restrict (and give information about) the intended behavior of a concrete implementation. The `IntAndAdd` implementation describes a concrete, external C++ API, and `IntAndAdd_models_Semigroup` establishes a modeling relation between `IntAndAdd` and the `Semigroup` concept.

Listing 1.2. A satisfaction relation in Magnolia.

```
implementation IntAndAdd = external C++ base.int_impl {
  type int;
  function add(i1: int, i2: int): int;
}

satisfaction IntAndAdd_models_Semigroup =
  IntAndAdd models Semigroup[ T => int, bop => add ];
```

As we declare modeling relations between concrete implementations and abstract specifications (or between abstract specifications), we build a database of knowledge about our Magnolia code. There are many ways in which this database of knowledge can be used, e.g., axioms can be used as unit tests [2], for formal verification of source code, or to produce rewriting rules (e.g. for optimizations)—much like in the Haskell GHC compiler [6].

Optimizations are typically implemented as passes in layered, but monolithic compilers. Rewriting rules derived from the specifications can however also be used, e.g., to perform arbitrary semantic-preserving transformations on the source code—i.e. for refactoring. Oftentimes, refactoring is done manually, at the IDE level.

The relevance of axiom-based rewriting outside of the automatic optimization case motivate the design of an extensible, modular ecosystem for Magnolia—in order to maximize code reusability.

2 Design of the Ecosystem

At the core of the ecosystem is the "monolithic" chunk of the Magnolia compiler. It has three parts: first, a parser. Second, a checker that simultaneously ensures that the parsed Abstract Syntax Tree (AST) is consistent and well-typed, and adds annotations to the AST. And third, a code generator for each available backend (currently C++, and Python). The fully annotated AST produced by the checker can be serialized, and serves as a common input format for the other plugins in the ecosystem (e.g. an IDE, a pretty printer, verification tools, and so on).

The database of knowledge contained in the fully annotated AST can then be fully exploited. Source code transformations are implemented as fully-annotated-AST to fully-annotated-AST transformations, ensuring their composability and that their output can be used by the other plugins in the ecosystem. Optimizations expressible in the form of axiom-based rewrites (and other semantic-based rewrites) are such transformations. Syntactic-level source transformations based on Syntactic Theory Functors [5] are also of interest. The ecosystem includes

plugins for both semantic and syntactic transformations. By pretty printing the AST resulting from the transformations, it can be fed back to the monolithic chunk of the compiler: the full Magnolia compiler is not a monolith, but instead a collection of small interconnected units—in a microservice-like fashion. Thus, as the ecosystem grows organically, so does the set of features of the compiler.

The experimental monolithic chunk of the Magnolia compiler is available at `https://github.com/magnolia-lang/magnolia-lang`. The development of an IDE for Magnolia is currently underway.

References

1. Bagge, A.H.: Constructs & Concepts: Language Design for Flexibility and Reliability. Ph.D. thesis, Research School in Information and Communication Technology, Department of Informatics, University of Bergen, Norway, PB 7803, 5020 Bergen, Norway (2009), `http://www.ii.uib.no/~anya/phd/`
2. Bagge, A.H., David, V., Haveraaen, M.: Testing with axioms in C++ 2011. Journal of Object Technology **10**, 10:1–32 (2011). https://doi.org/10.5381/jot.2011.10.1.a10, `https://doi.org/10.5381/jot.2011.10.1.a10`
3. Goguen, J.A., Burstall, R.M.: Introducing institutions. In: Clarke, E., Kozen, D. (eds.) Logics of Programs. pp. 221–256. Springer Berlin Heidelberg, Berlin, Heidelberg (1984)
4. Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Reis, G.D., Lumsdaine, A.: Concepts: linguistic support for generic programming in C++. OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications pp. 291–310 (2006). https://doi.org/http://doi.acm.org/10.1145/1167473.1167499
5. Haveraaen, M., Roggenbach, M.: Specifying with syntactic theory functors. Journal of Logical and Algebraic Methods in Programming **113**, 100543 (2020). https://doi.org/https://doi.org/10.1016/j.jlamp.2020.100543, `https://www.sciencedirect.com/science/article/pii/S2352220820300286`
6. Peyton Jones, S., Tolmach, A., Hoare, T.: Playing by the rules: rewriting as a practical optimisation technique in ghc. In: 2001 Haskell Workshop. ACM SIGPLAN (September 2001), `https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/`