Modular and Generic Programming with InterpreterLib

Philip Weaver, Garrin Kimmell, Nicolas Frisby, Perry Alexander
Information and Telecommunication Technology Center
Electrical Engineering and Computer Science Department
The University of Kansas, 2335 Irving Hill Road, Lawrence, KS 66045
{pweaver,kimmell,nfrisby,alex}@ittc.ku.edu

ABSTRACT
Modular monadic semantics (MMS) is a well-known technique for structuring modular denotational semantic definitions. Families of language constructs are independently defined using syntactic functors and semantic algebras that can be combined in a mix-and-match fashion to create complete language definitions. We introduce InterpreterLib, a Haskell library that implements and extends MMS techniques for writing composable analyses. In addition to modular analyses composition, InterpreterLib provides algebra combinator functions, explicit algebra semantics, preprocessors for boilerplate generation and generic programming techniques adapted to language analysis. The key benefits of these features are reliability, increased code reuse via modularity and the ability to rapidly retarget component analyses.

Categories and Subject Descriptors
D.2.13 [Software Engineering]: Reusable Software - Reusable libraries

General Terms
Languages

1. INTRODUCTION
Developing tools to interpret or analyze programming languages is a difficult and time-consuming task. This is especially true when the underlying target language is evolving alongside the tools, as is often the case with new or experimental languages. Developers must deal readily with a shifting target, while minimizing the effort required to adapt tools. Two key factors in successfully navigating these challenges are well known software engineering principles: modularizing design to localize modifications and abstracting common computational patterns into reusable components. In this paper, we describe InterpreterLib, a system for writing language analyses founded on these two principles that draws heavily from modular monadic semantics and generic programming.

Although originally developed for traditional language interpreters, InterpreterLib now targets developing tools for numerous software engineering tasks. Specifically, InterpreterLib has been successfully applied to developing static analysis tools, type checkers, VHDL synthesis tools, and symbolic simulation tools. Current applications include synthesis of software defined radios in C++ and VHDL, synthesis of models for software model checking, and verification of abstraction and concretization functions.

InterpreterLib is written in Haskell and combines these features into a single framework that provides: (i) modular composition of syntactic elements and corresponding semantics; (ii) generic definitions of analyses, applicable to a wide variety of constructs; (iii) high-level operators for combining dissimilar analyses; and (iv) automated generation of requisite boilerplate code. In the remainder of this paper, we briefly describe these features and offer preliminary metrics calculated from experience using InterpreterLib.

2. BACKGROUND
InterpreterLib is firmly rooted in the framework of Modular Monadic semantics (MMS). In this section we give a brief overview of the framework; a more comprehensive overview of MMS can be found in the numerous background references [3, 12, 2, 5].

MMS is founded in the observation that all language analyses have the same general structure: a function mapping from an Abstract Syntax Tree (AST) data structure to some value space that represents the result of the analysis. There are four major components to this general structure, and MMS achieves flexibility by modularizing each of these in turn. The components include syntactic functors, that capture the abstract syntax of the language; composible value spaces represent the result of the analyses; semantic algebras are the functions mapping syntactic functors (AST) to value spaces; and finally monad transformers that represent the computational effects generated by the semantic algebras.

Syntactic functors are non-recursive forms of inductive datatypes as typically defined in a functional language. A single syntactic functor represents a family of closely related language constructs, each of which is represented by a constructor for the functor datatype. Two independent syntactic functors can be combined using a sum functor. By making the functors non-recursive, it is possible to add – or remove – individual functors from a sum of functors. The least fixed-point of the sum of functors is semantically equivalent to a
single, inductively defined datatype that includes all of the constructors contained in the component functors.

Composable value spaces are constructed as the sum (co-product) of component value spaces. In the MMS framework, rather than relying on a fixed value space, analyses use injection and projection functions that allow component value spaces to be embedded into an overall value space. Consequently, the semantic algebras described below need only be concerned with the specific language constructs (syntactic functor) and value space for a particular family of constructs.

Semantic algebras relate syntactic functors to value spaces, defining an analysis semantics. A semantic algebra for a functor $F$ and value space $v$ will have the type $F 	o v$. The value space $v$ is called the carrier of the algebra. Similarly, to the composition of syntactic functors and value spaces, InterpreterLib contains an operator that allows two semantic algebras, for two different functors, to be combined into a single algebra.

Finally, a semantic algebra will often use computational side effects during a language analysis. Because the framework is hosted in a pure language, these effects are captured as monadic computations. Consequently, an analysis that includes effects will have a monadic carrier. Two different semantic algebras for the same analyses may require two different monads. Monad transformers provide the ability to represent a monad that captures the effects of both monads.

### 3. GENERICS AND COMBINATORS

In addition to reuse provided by composite syntactic functors and semantics algebras, InterpreterLib provides generic programming and algebra combinators to support reusing entire code analyses. A generic algebra can be applied to many different datatypes without alteration. Algebra combinators enable cooperation between individual analyses to implement complex synthesis and analysis capabilities.

#### 3.1 Updateable Generic Algebras

Generic programming is a class of techniques for concise definitions of functions that remain complete even when datatypes are changed [11]. This robustness results in highly adaptable and modular code, introducing significant code reuse opportunities. A particular approach to generic programming, updateable algebras [11], recognizes that the definition of a function is uniform for most constructors with a few exceptional cases. The behavior of the generic function for the exceptional cases can be updated to perform the special functionality instead of the generic default.

A collection of generic algebras provided by InterpreterLib that capture common default behavior found in language analyses serve as the starting point for defining generic analyses. The generic algebra handles the majority of the syntactic constructs (including those not yet defined), and a programmer uses an update operator to specialize the default behavior for specific functors.

#### 3.2 Algebra Combinators

Whereas updateable generic algebras enable the definition of widely-applicable analyses, algebra combinators allow the definition of a new algebra as the composition of existing algebras. The combinators provide a natural expression of dependencies between analyses. The two algebra combinators currently provided by InterpreterLib are sequence, for the chaining of analyses, and switch for handling modes represented by multiple algebras within a single analysis activity.

**Sequence** Language processors often reuse the results of static analyses in the rest of a tool flow. Fundamental analyses, such as free variable analysis or type-checking/inference, may be used later during more sophisticated analyses, allowing significant reuse of the base analysis. Sequencing one algebra after another provides the results of the first algebra to the second algebra at each location within the AST. The combinator can combine an arbitrary number of static analyses.

**Switch** An analysis may be modal, where contextual information dictates which of two algebras (modes) is applicable. This behavior is captured using a switch algebra combinator. The operation of each mode is described with an algebra, all of which are combined into a single algebra that switches between the alternatives. All algebras share a single notion of dynamic context called the switch that each manipulates to transfer analytic control to a different mode. This combinator combines two algebras over common function and carriers into a single algebra.

### 4. ALGC

Integrating a language into the InterpreterLib framework entails a significant initial cost, as each syntactic functor must be accompanied by a large amount of boilerplate code, most of which is Haskell class instance declarations. For this reason, InterpreterLib includes a preprocessor, AlgC, that generates all necessary boilerplate code from simple functor specifications written in a domain-specific functor specification language.

AlgC generates about 20 to 30 lines of code for each language construct, including: (i) a Haskell datatype; (ii) instances of seven type classes necessary to integrate the functors into the InterpreterLib framework; and (iii) a “smart” constructor for each construct. The smart constructor allows the construct to be inserted into any composite AST type.

AlgC can also process ASTs defined directly as Haskell datatypes, instead of the special functor description language. When processing Haskell as the source abstract syntax, AlgC automatically generates a `convert` function that translates abstract syntax from the recursive form to a functorial representation. This makes it easier to port existing languages to InterpreterLib. The user simply calls `convert` on the recursive datatype and then applies tools written in InterpreterLib.

### 5. EXPERIMENTAL RESULTS

InterpreterLib provides a range of facilities for constructing modular language processing tools. A principal feature of the framework is that it leads to considerable reuse and reduction of code necessary to implement the tools.

To demonstrate these results, in this section we present a lines-of-code (LOC) analysis of example applications, both for small (contrived) languages as well as for larger-scale tools supporting analysis of the Rosetta [1] specification language. While admittedly a debatable metric for calculating reuse, these numbers indicate reductions in user-written code often greater than an order of magnitude.

#### 5.1 AlgC
The InterpreterLib framework relies on several Haskell type classes that each syntactic functor must implement. Currently, AlgC generates instances for seven classes. It is possible to use some portions of the InterpreterLib framework having only implemented a subset of these seven classes, but to utilize the full extent of the library all are required. The critical attribute that determines the amount of programming effort necessary to implement these classes without using AlgC is the number of constructors for each syntactic functor. In our experience using InterpreterLib over a collection of common language constructs, we observe an approximately 10-15 times size increase from the number of constructors in an original functor specification (3 LOC per functor) to the generated boilerplate class instances (averaging 40 LOC per functor).

When using the features of AlgC that generate the InterpreterLib boilerplate from recursive Haskell datatype declarations, the code expansion becomes even more significant, due to the generation of code to convert the recursive form to its non-recursive equivalent. For example, the Rosetta AST includes 60 data constructors, separated into 13 functors. The Haskell representation of this AST measures 60 lines of code. After processing by AlgC, the generated code—excluding the 6 class instances mentioned above and the conversion function—is 1356 lines of code, over 22 times larger.

5.2 Generic Programming

The two-phase approach to generic programming—where generic behavior is first selected, and then specific cases are updated with specialized behavior—inherently involves a reduction in the LOC metric, since most constructors are handled by a generic algebra. Here we present the LOC metrics for a number of analyses from the Rosetta simulator. Though we list concrete measurements, the software engineering benefit is difficult to quantify because we cannot predict the number of future constructors handled by the generic algebra.

In the simulator, an elaboration step replaces one of the 13 functors in the Rosetta AST generated by AlgC with a functor with 3 more constructors. Thus, most functions handle 63 constructors over 13 functors. In Table 1, the Estimated LOC column corresponds to the number of constructors dealt with by the generic algebra. The estimate is based on the conservative assumption that, when written manually, handling these default cases would result in one LOC per constructor. The Actual LOC column corresponds to the number of invocations (one LOC per invocation) of the generic algebra to cover those cases. These figures do not include LOC counts for specialized behavior, as those definitions are the same with or without using the generic programming. The analyses listed include common AST query operations, as well as Rosetta-specific concerns.

5.3 Algebra Combinators

AlgC’s generative features and InterpreterLib’s generic programming features reduce the amount of hand-written code necessary to build language processing tools. Structuring analyses as collections of smaller algebras composed using algebra combinators allows wholesale reuse of those component algebras. The use of algebra combinators in InterpreterLib is described in depth in [14]; we use that work to provide the basis for measuring reuse due to algebra combinators.

The first example uses a switch algebra combinator to add Lisp-style metaprogramming to a simple functional language extended with products, sums, boolean and arithmetic expressions. An interpretation semantics (29 LOC) is combined with a semantics for metaprogramming (17 LOC). Combining the two required an addition 20 lines of “glue” code to yield a semantics for the original language extended with metaprogramming facilities. The interpretation semantics was reused without modification. Furthermore, given an interpretation semantics for a different language, both the metaprogramming semantics and the glue code can be reused unmodified.

The second example uses sequence algebra combinator to use the results of a type checking semantics for synthesizing FPGA netlists from a language similar to the simple language from the metaprogramming example. Without using the seqalg, type checking logic must be duplicated within the synthesis algebra. In contrast, when using seqalg to combine an independently-defined type checking algebra (31 LOC) with a type-directed synthesis algebra (41 LOC), the type checking algebra’s definition can be reused without modification and the synthesis algebra is not polluted with the details of type checking.

6. RELATED WORK

Many of the generic programming techniques employed in the development of InterpreterLib rely on the initial algebra semantics of datatypes [4]. This defines a formal semantics for inductive types in a category-theoretic setting. In addition to providing a mathematical interpretation of inductive datatypes, this work led to significant programming advantages. Large classes of algorithms can be described generically based on the shape of the datatype’s functorial representation characterized using a small collection of primitive shape functors (constants, sums, products, and finite exponents). Datatypes captured with this collection of primitive functors are called polynomial datatypes. Although InterpreterLib focuses primarily on language analyses, the techniques are applicable to any primitive-recursive function defined over polynomial datatypes.

PolyP [6] is a generic programming extension for Haskell that defines a language for generic functions based on primitive shape functors. PolyP uses a preprocessor convert these generic functions into Haskell. PolyP also defines a Regular type class that allows the functor shape to be extracted from regular Haskell datatypes. The preprocessor generates the instances for the Regular class automatically. The AlgC preprocessor included in InterpreterLib performs a similar function.

Table 1: Generic Programming Measurements

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Estimated LOC</th>
<th>Actual LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elaboration</td>
<td>53</td>
<td>3</td>
</tr>
<tr>
<td>Names</td>
<td>47</td>
<td>5</td>
</tr>
<tr>
<td>Vars in Scope</td>
<td>60</td>
<td>3</td>
</tr>
<tr>
<td>Exported Vars</td>
<td>58</td>
<td>2</td>
</tr>
<tr>
<td>Variable Kinds</td>
<td>58</td>
<td>2</td>
</tr>
<tr>
<td>Stateful Vars</td>
<td>59</td>
<td>3</td>
</tr>
</tbody>
</table>


The initial algebra approach to generic programming utilized in the above systems is prevalent, but suffers from the disadvantage that it draws a distinction between the recursive form of datatypes common in functional languages and the non-recursive functor-based representation. An alternative approach [9], uses a dynamic type-safe cast to allow generic algorithms to be written over recursive datatypes. Although this approach has the advantage of simplicity, it lacks the uniformity and formal grounding of the initial algebra semantics framework.

Stratego and Strafunksni are built around the “Scrap your Boilerplate” approach [9, 10, 13] and inherit both its advantages and shortcomings. Those frameworks include an operator similar to the sequence algebra combinator as demonstrated in this paper. Though not demonstrated in here, sequence algebra can hide the monadic features of a carrier in the first algebra from a second algebra without violating the monadic encapsulation. This feature enables more complex algebra composition.

The framework of modular monadic semantics simplifies schemes for defining and composing interpreters [3, 12]. In particular, the Language Prototyping System (LPS) [7] provides a means for developing modular monadic interpreters in a manner quite similar to InterpreterLib, yet lacking the capabilities for algebra composition, generic algebras or automatic boilerplate synthesis.

7. CONCLUSIONS

InterpreterLib is a framework for defining modular language processors in Haskell. InterpreterLib combines techniques from modular monadic semantics and generic programming to simplify the construction of such tools. A major advantage of the framework is that it allows processing components – algebras – to be defined in a modular and extensible fashion. Algebra combinators allow these semantic algebras to be combined both sequentially using the sequence algebra combinator and in a parallel (either/or) fashion using the switch algebra combinator.

Frequently, language analysis will operate on only a specific portion of large AST structures. InterpreterLib’s generic programming features allow these analyses to be defined as updates to a rote generic functionality. Furthermore, InterpreterLib provides features for updating algebraic pieceswise, facilitating significant reuse of existing algebra definitions.

To implement these features, InterpreterLib relies on the Haskell type class system to provide interfaces for generic programming and algebra manipulation features. This has the potential to burden users with generation of large amounts of support code to provide instances for InterpreterLib classes. To alleviate this problem, InterpreterLib provides the AlgC preprocessor to automatically generate boilerplate code, reducing both programmer effort and potential for error. InterpreterLib is used extensively in tools for the Rosetta system-level design language [1]. The Rosetta language has several features, such as support for heterogeneous computational models and formal analysis, that demand tools be constructed to allow a high degree of assurance that they are constructed correctly in addition to being easily repurposed for different analysis tasks. Our experience in developing these tools within the framework has indicated that InterpreterLib is successful in supporting both of these aims.

8. REFERENCES