Technical Report

Composing Specifications Using Algebra Combinators

Garrin Kimmell, Jennifer Streb, Ed Komp
and Perry Alexander

ITTC-FY2006-TR-30150-06

March 2006

Project Sponsor:
National Science Foundation

Abstract

The need to understand effects of cross-cutting concerns defines the essence of systems-level design. Understanding the impacts of local design decisions on global requirements such as power consumption or security is mandatory for constructing correct systems. Unfortunately, domain specific models may be defined using different semantics making analysis difficulty. We define an *algebra combinator* that provide semantics for model composition. Given two models defined over a common abstract syntax, an algebra combinator defines a single model that embodies the composition of those specifications. Such composite models can then be used to understand the interaction of models from the original specification domains.

1 Introduction

Systems-level design places additional burdens on designers to understand the interaction of seemingly independent concerns in the realization of a final system. Oftentimes, the vocabulary used to express a design differs across the various (seemingly independent) requirements of the system. There are two interrelated issues that are significant when performing analysis of system-level requirements. First, the *model of computation* will vary, depending on the behavior being specified. For example, when specifying the functional behavior of a synchronous circuit lends itself to a discrete state-based model of computation, while the specification of an analog circuit will more naturally be expressed using a continuous-time model of computation. It is important to be able to perform analysis of models expressed in these varied models of computation.

Secondly, the semantics of a requirement specification will largely depend on the system properties being modeled. The “meaning” of an expression is different if it is being used to specify functional behavior – where the concern is the value being computed – than the meaning of the expression when used to analyze power requirements of the system.

In this work, we describe a method for resolving these issues by systematic development of language semantics. We formalize this techniques by structuring the semantics using monads[1, 2] to model varied “notions of computation”. Additionally, we use a technique for extending languages in modular fashion, which allows the introduction of new specification “vocabulary” without a wholesale reformulation of language semantics. Finally, we introduce the notion of an *algebra combinator* to facilitate the analysis of specifications across domain boundaries.

We apply our approach to the analysis of the Rosetta [3, 4] specification language. Rosetta is a language for specifying heterogeneous systems structured around *domains*, which resemble models of computation. Domains are structured in a hierarchy, with an extension relation between the domains. Figure 1

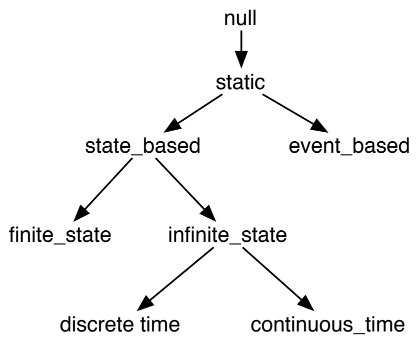

Figure 1: Rosetta domain hierarchy

shows a portion of the basic Rosetta domain hierarchy. The `static` domain corresponds to pure computations without side effects. Traveling down the domain hierarchy we encounter increasingly sophisticated models of computation.

In our analysis framework, a domain manifests itself in three ways:

1. A model of computation
2. A semantics, mapping expressions to computations of the chosen model
3. A set of syntactic elements, used for accessing the features of the computational model

To implement a domain analysis tool, we utilize techniques for defining language semantics known as *modular monadic semantics*[5, 6]. The above characterization of a domain fits well into this work. *Monads* provide the realization of computational models. Language semantics are manifested as *semantic algebras* over syntactic elements present in the domain. The inheritance hierarchy present in the domain hierarchy is manifested as syntactic and associated semantic extension. By providing a structured language extension mechanism, modular monadic semantics allows the reuse of language semantics: semantics for a parent domain can be used as the basis for the semantics of an extended domain by adding basic computational features (monads) and vocabulary (syntax).

2 Modular Monadic Semantics

Denotational semantics is a semantic specification technique that maps terms from a syntactic domain into a particular semantic domain. Constructs with effects require additional parameters to the denotation function. This results in a significant limitation of denotational semantics, in that semantics are non-compositional. Adding a new language construct requires a complete rewriting of the denotational specification. Traditional denotational semantics is unattractive for modeling analysis semantics for Rosetta because of the above limitations. Various Rosetta domains model computation using different effects, as well as introduce new syntax for accessing these effects.

We can use *modular monadic semantics* to resolve these limitations. First, we add computational effects to the semantic domain, rather than embedding it in the denotation function, by making the semantic domain monadic. We structure the abstract syntax data type as the least fixpoint of a collection of independent language constructs, expressed as functors. This provides the ability to extend the syntax of the language. Denotations are expressed as *algebras* over the abstract syntax, mapping from syntax to the monadic semantic domain. These algebras are composed in a systematic fashion to create a semantics for a complete language, within a particular domain. Finally, we will use *algebra combinators* to compose varied semantics for different domains to facilitate heterogeneous analysis.

2.1 Monads

A monad [2, 1] is a construction from category theory that can be used to model a wide variety of computational effects. A monad is a triple $\langle\ T, unit, bind \rangle$ where T is a type constructor for computations, parameterized over the type of the *value* the computation will result in. The *unit* morphism lifts values, of type a, to computations of type $T\ a$. This can be understood as a trivial computation, that has no effect. Finally, *bind* lifts functions from values to computations into functions from computations to computations. The *bind* morphism serves to sequence computations, yielding the effectful behavior of monadic computations. In addition to the above signature, a monad must satsify a collection of algebraic laws, which we elide.

Using this simple formulation, we can construct a monad which models the lambda calculus. The type constructor T is a identity functor, *unit* is the identity function, and *bind* is function application. More complex monads, modeling various compuational effects, can be defined by adding a collection of non-proper morphisms to the signature. It is these morphisms that provide access to the features of the computational model.

Consider the state monad, which models imperative computations. The type constructor T is modeled as function mapping a state parameter to a pair of state and value. The *unit* morphism maps values to functions of this type, where the state is not utilized. The *bind* morphism allows the “threading” of state through computations. Figure 2 gives an example of this monad, implemented in the Haskell [7] programming language. In addition to the *bind* and *unit* functions, we introduce the non-proper *get* and *put* morphisms, which allows the manipulation of the state.

$$\begin{array}{l}
\textbf{type}\ \mathit{StateMonad}\ s\ a = s \rightarrow (s, a) \\
\mathit{unit}\ \mathit{val} = \lambda s \rightarrow (s, \mathit{val}) \\
m\ \text{`}\mathit{bind}\text{`}\ f = \lambda s \rightarrow \textbf{let}\ (s', a) = m\ s\ \textbf{in}\ f\ a\ s' \\
\mathit{get} :: \mathit{StateMonad}\ s\ s \\
\mathit{get} = \lambda s \rightarrow (s, s) \\
\mathit{put} :: s \rightarrow \mathit{StateMonad}\ s\ () \\
\mathit{put}\ s' = \lambda\_ \rightarrow (s', ())
\end{array}$$

Figure 2: The State Monad in Haskell

In addition to the state monad, a wide range of monads, capturing a wide range of effects, have been identified. These include input/output, non-determinism, read- and write-only state, exception handling, continuations, and concurrent features. One difficulty when constructing monadic computations is that monads, in general, do not compose. However, monad transformers [6] can be used to independently capture features from specific computational effects, and then combine to create a monad that has all of the desired effects.

To illustrate the use of our approach, we take a small subset of the complete Rosetta expression language. This subset is described in Figure 3. It includes arithmetic operations, named values introduced with `let` bindings, and an `if` construct.

expr	:=	letexpr | ifexpr | arithexpr
letexpr	:=	**let** v **=** expr **in** expr
		v
arithexpr	:=	expr op expr | **N**
ifexpr	:=	**if** expr **then** expr **else** expr
		true
		false
v	:=	**a** | **b** | **c** | ...
op	:=	**+** | **-** | ***** | **/**

Figure 3: Rosetta Language Subset

2.2 Composing Syntax

While the grammar in figure 3 can be directly translated into a recursive data type, we separate the expression elements into three different data types, reflecting the orthogonal nature of the different language constructs. These three types constructs are `let` bindings (with variables), `if` expressions, and arithmetic operations. The intuition behind the partitioning is that let expressions require a read-only state monadic effect. The arithmetic and boolean expressions, while exhibiting no particular computational effect, have separate semantic domains, with arithmetic expressions yielding numerical values and `if` expressions resulting in boolean values.

The Haskell encoding of the independent abstract syntax tree (AST) constructs is shown in Figure 4. Each construct family is specified as a type, with AST elements corresponding to constructors for that particular type. The AST types are written non-recursively, with a type parameter x. This allows the addition of additional syntax without necessitating the redefinition of the term data type. The *Sum* type allows the independent AST types to be "glued" together. The *Fix* type calculates the fixed point of a non-recursively defined term. This fixed point, defined using the *Expr* type synonym, is isomorphic to a recursively defined AST containing all of the component constructs.

```
data Arith x = Add x x | Sub x x | Mul x x | Div x x | Num Int
data If x = If x x x | Tru | Fls
data Let x = Let String x x | Var String

data Sum f g x = InjL (f x) | InjR (g x)
data Fix f = Mu (f (Fix f))

type Expr = Fix (Sum Arith (Sum If Let))
```

Figure 4: Independent Abstract Syntax

2.3 Composing Semantics

Having defined a mechanism for independently representing syntactic constructs, we can now assign a meaning to the constructs by defining *algebras*. Each algebra has the form $F\ a \rightarrow a$, where F is the (non-recursive) term type, and a is the semantic domain of the algebra. To simplify the definitions, we use the Haskell **do** notation, which is a syntactic sugar around the monadic *bind* and *unit* constructs. For brevity, we omit all arithmetic operations except for addition. The others are defined in the same way, with the appropriate syntax and operator substitution. The *Let* form uses the *Reader* monad, which provides read-only state, appropriate for modeling an environment for let bindings. The two non-proper morphisms *lookupEnv* and *extendEnv* allow access to the effects of the *Reader* monad. The ϕ_{sum} algebra allows the composition of indvidual algebras, as shown in the ϕ binding.

The particular semantics defined in Figure 5 targets booleans and integers as a semantic domain. However, ϕ_{arith} only requires that integers be in the domain, ϕ_{if} only requires booleans, and ϕ_{let} is purely for compuational effect and places no constraint on the semantic domain.

These are not the only possible denotations for these language constructs. For example, rather than embedding the semantics as an interpreter in the Haskell

```
$\phi_{arith}$ (Add x y) = do xv ← x
                              yv ← y
                              return (xv + yv)
$\phi_{arith}$ (Num x) = return x

$\phi_{if}$ (If pred th els) = do pv ← pred
                                 if pv then th else els
$\phi_{if}$ Tru = return True
$\phi_{if}$ Fls = return False

$\phi_{let}$ (Let n binding body) = do bv ← binding
                                      extendEnv n bv body
$\phi_{let}$ (Var x) = lookupEnv x

$\phi = \phi_{sum}\ \phi_{arith}\ (\phi_{sum}\ \phi_{if}\ \phi_{let})$

$\phi_{sum}\ \phi_l\ \phi_r$ (InjL x) = $\phi_l$ x
$\phi_{sum}\ \phi_l\ \phi_r$ (InjR x) = $\phi_r$ x
```

Figure 5: Semantic Algebras

metalanguage, we could have easily defined a different semantics (with a different semantic domain) to define a compiler for the same language constructs[8].

In this section we've describe the background for implementing analysis semantics using modular monadic semantics. Monads allow us to model computational effects. We can extend a language by adding new constructs, without disturbing the existing syntactic and semantic definitions. Finally, we can retarget the semantics for a particular construct for various semantic domains. In the next section, we will demonstrate how to use *algebra combinators* to perform heterogeneous analysis of models written using these constructs.

3 Combining Semantics

We can use the above approach to define compilation semantic to generate netlists for implementing these high-level constructs in hardware [9, 10, 11, 12]. This application gives motivation for cross-domain analysis of models. When implementing systems in hardware, we are often concerned with the power consumption of the system, in addition to the functional behavior.

To illustrate this technique, we utilize a simplistic power model, with circuit components modeled as black boxes which consume a given amount of power for each time they generate a value. Using this model we demonstrate the impact the implementation strategy, expressed as a compilation algebra to a circuit configuration, has on power consumption.

In our examples, we assume that the circuit model for `let` expressions and arithmetic remain constant, while our model for `if` expressions varies. When generating circuits for let expressions, we assume that we generate a single circuit for the binding, then reuse the calculated value throughout the body of the let expression. The power consumed by a variable reference is negligible, as the value has been pre-computed. In circuit terms, a variable reference is simply a wire from the binding circuit. Furthermore, arithmetic expressions generate a circuit by inlining the appropriate circuit for the given arithmetic operator. Integer constants consume a predefined amount of power. These power semantics are shown in Figure 6.

```
$\phi_{letpower}$ $(Let\ n\ b\ body) =$
  **do** $bpower \leftarrow b$
     $extendEnv\ n\ 0\ body$
$\phi_{letpower}$ $(Var\ x) = lookupEnv\ x$

$\phi_{arithpower}$ $(Add\ x\ y) =$
  **do** $xpower \leftarrow x$
     $ypower \leftarrow y$
     $return\ (xpower + ypower)$
$\phi_{arithpower}$ $(Num\ n) = return\ constantPower$
```

Figure 6: Let and Arithmetic Power Semantics

3.1 Static Power Semantics

The first analysis we construct is simplified greatly because the functional simulation and the power analysis are dynamically independent. In Figure 7, we see that the predicate and both branches are to be evaluated in parallel. A multiplexer is used to determine which of the branch outputs is output from the complete circuit. With this configuration, the power consumed by the circuit is independent of the value calculated by the predicate. This design decision yields an potential increase in circuit speed, since there is no data dependency between the predicate circuit and the circuits of each branch.

Figure 8 shows the power semantics for the parallel if expression. Note that the power consumed by this circuit is the sum of the component circuit, along with the additional power overhead of the multiplexer.

This simple power model allows us to generate a rough estimation of power consumption independently of values generated by the circuit. It is trivial to combine these power semantics with the value semantics, but this is not a particularly compelling use of cross-domain analysis. We shall see that with a

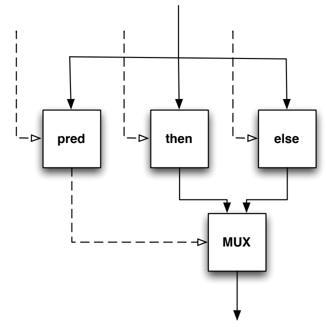

Figure 7: A parallel `if` statement.

```
$\phi_{parallelifpower}$ $(If\ pred\ thn\ els) =$
  **do** $ppower \leftarrow pred$
     $tpower \leftarrow thn$
     $epower \leftarrow els$
     $return\ (ppower + tpower + epower + muxpower)$
$\phi_{parallelifpower}$ $Tru = truPower$
$\phi_{parallelifpower}$ $Fls = flsPower$
```

Figure 8: Parallel `if` power semantics

more sophisticated power model and different circuit compilation semantics, the analysis of power semantics cannot be carried out independently of the value analysis.

3.2 A Dynamic Power Model

The compilation scheme for if expressions shown in Figure 8 is not the only possible circuit configuration. An alternative is demonstrated in Figure 9. In this model, the output of the predicate is used as a control circuit for the two `if` branches. This introduces a control dependency between predicate and each branch, potentially causing a performance decrease for the circuit. Given our general power semantics, where circuits are black boxes which only consume power when activated, this configuration will result in a power savings since only one of the two branches is activated.

We can construct a power model for this circuit by using a *non-determinism* monad. This monad allows a computation to return a list of values, representing each value the computation could potentially result in. Figure 10 shows how the power semantics would use this computational model. The *amb* morphism exposes the multiple return values behavior of this monad.

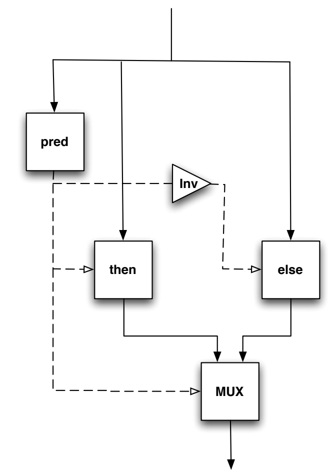

Figure 9: A sequential `if` statement

```
$\phi_{nondetifpower}$ (*If pred thn els*) =
  **do** *ppower* $\leftarrow$ *pred*
     *tpower* $\leftarrow$ *thn*
     *epower* $\leftarrow$ *els*
     *amb* (*ppower* + *tpower* + *invpower*)
          (*ppower* + *epower* + *invpower*)
$\phi_{nondetifpower}$ *Tru* = *return truPower*
$\phi_{nondetifpower}$ *Tru* = *return flsPower*
```

Figure 10: Nondeterministic Power

3.3 Composing Functional and Dynamic Power Semantics

The nondeterministic power semantics for the if circuit captures the power consumption along every possible execution trace of the circuit. While this gives a complete model of the power consumption, it is not possible to generate the power profile for a single execution trace.

The difficulty with combining the evaluation semantics and power consumption semantics is that they are expressed as separate computation, in separate monads. Assuming the value semantics are expressed in m, and the power semantics in monad n, we would like to generate a single monad $m+n$, which has both the effects of m and n. Furthermore, we need to the combined semantics to generate not only power consumption estimations, but we also must return the results of the value semantics, because it is part of an input to the power semantics.

A first attempt at accomplishing this results in the *pairAlg* function. However, this has two large problems. First, we need the results of applying the value semantics to the predicate as an input to the power semantics. Secondly, it is not possible to escape from a monadic computation, so it is not possible to extract a result from the value computation and "insert" it into the power computation.

```
pairAlg phi1 phi2 term = do x ← phi1 term
                            y ← phi2 term
                            return (x, y)
```

To resolve this issue, we add an extra monadic computation around the two analysis monads. This gives us a final computation type i $(m\ value, n\ power)$, where i is the "interaction monad". Because the analysis for value and power semantics proceed in lock-step, we maintain in the interaction monad a labeling for each dynamic application of the semantic algebra. The two semantic algebras communicate using this labeling, through a channel in the interaction monad.

With this scheme, we can generate the interacting algebra behavior we desire. The functions *writeDynamicValue* and *readDynamicValue* are used to communicate between the two algebras.

```
interact_if_value (If pred thn els) =
  do pvalue ← pred
     writeDynamicValue pvalue
     return ($\phi_{if}$ (If (return pvalue) thn els))

interact_if_power (If pred thn els) =
  do pvalue ← readDynamicValue
     if pvalue
       then
         ($\phi_{parallelifpower}$ (If pred then (return 0)))
       else
         ($\phi_{parallelifpower}$ (If pred (return 0) els))

interact s1 s2 term =
  do m ← s1 term
     n ← s2 term
     return (m, n)

interact_if_semantics = interact interact_if_value interact_if_power
```

With this construction we can perform cross-domain analysis of interacting models in a systematic manner. The interaction monad is essentially a state monad, with *writeDynamicValue* and *readDynamicValue* as convenience functions for accessing the state. The interaction semantics defined above are ad-hoc, because heterogeneous interaction is in general an ad-hoc construction. However, our systematic approach to generating these interactions is useful for guiding the definition of the ad-hoc interactions.

4 Related Work

Monads were first identified by Moggi[1, 2] as a mechanism for modeling computation in a mathematical setting. Later work [13, 14, 15] demonstrated the applicability of monads as a programming tool to obtain non-pure computation in a pure language. It is this early work on monads that forms the basis for our representation of models of computation.

The specific application of monads to structuring language semantics was introduced by Espinosa[16], and was later extended to monad transformers to solve the problem of non-composing monads[5, 6]. Defining a denotation as an algebra[17, 18] extended the earlier work to allow compositional and extensible syntax. Using this technique, a large range of common language constructs can be modeled independently, then combined into a complete language[19, 20, 21]. Exposing the algebras as first-class data objects[22] allows the language semantics to be manipulated directly [23].

Goodman describes the use of monads for the purpose of animating Z specifications [24]. This work differs from ours in that it provides a translation from Z into an executable Haskell program, written in monadic style, manually. In contrast, we use monads to structure the analysis semantics of Rosetta itself. Abdallah et. al[25] demonstrated the systematic refinement of Z specifications to an executable prototype in functional language in a non-monadic fashion. While the focus in both these works is on modeling state-transition systems, our approach using monads allows us to analyze specifications written using other computational models.

Implementing systems by factoring cross-cutting concerns and introducing them in a regular manner is the motivation behind aspect-oriented programming (AOP). Aspect languages can be characterized as a meta-programming language. Cross-cutting concerns are implemented in the aspect language, and "woven in" the system using augmented semantics of the object language. The semantics of aspect languages[26] are manifested as manipulation of the underlying language, in much the same manner as our algebra combinators.

The power model we use as an example in section 3 is very simplistic. More accurate power models are available which base power estimation on switching activity in CMOS circuits [27, 28]. These models are expressed as continuous functions, most appropriately modeled in the continuous domain from Figure 1.

5 Conclusions

In this paper we have demonstrated a framework for the analysis of Rosetta specifications. Rosetta is structured in a hierarchy of domains, offering vocabularies and computational models to be chosen as appropriate for a particular specification task. We structure our analysis framework as a semantics for a collection of languages, each corresponding to a Rosetta domain. A domain semantics consists of a model of computation, manifested as a monad in the semantics, as well as domain-specific syntax.

To facilitate heterogeneous analysis, we combine specification semantics using algebra combinators. First, we demonstrate that single specification, when interpreted with different semantics, allows us to analyze different properties of a system. Secondly, we use the same technique to combine analysis semantics in a dependent manner.

5.1 Future Work

While this framework allows the systematic generation of analysis tools using monads, the combination of semantics using algebra combinators places a significant burden on the tool developer to construct an appropriate monad for heterogeneous models. We plan to explore the use of more powerful computational effects [29, 30] for defining deriving these complex monads.

As noted above, the example power model from section 3 is very simplistic. We plan to extend this work to utilize more realistic activity-based models. This will result in more meaningful estimation results, as well as demonstrate the ability of the framework to bridge the gap across more disparate computational models.

References

[1] Moggi, E.: Notions of computation and monads. Information and Computation **93** (1991) 55–92

[2] Moggi, E.: An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Dept. of Computer Science, Edinburgh Univ. (1990)

[3] Alexander, P.: Multi-facetted design: The key to systems engineering. In: Proceedings of Forum on Design Languages (FDL-98). (1998)

[4] Kong, C.: Modular Semantics for Model-Oriented Design. Phd dissertation, The University of Kansas, Lawrence, KS USA (2004)

[5] Liang, S., Hudak, P.: Modular denotational semantics for compiler construction. In: Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping. Volume 1058., Springer-Verlag (1996) 219–234

[6] Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In ACM, ed.: Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages:

San Francisco, California, January 22–25, 1995, New York, NY, USA, ACM Press (1995) 333–343

[7] Jones, S.P.: Haskell 98 Language and Libraries. Cambridge University Press (2003)

[8] Harrison, W.L., Kamin, S.N.: Metacomputation-based compiler architecture. In: Mathematics of Program Construction. (2000) 213–229

[9] O'Donnell, J.: Generating netlists from executable circuit specifications in a pure functional language. In: Functional Programming, Glasgow 1992. Workshops in Computing, Springer-Verlag (1992) 178–194

[10] Mycroft, A., Sharp, R.: Hardware/software co-design using functional languages. In: Tools and Algorithms for Construction and Analysis of Systems. (2001) 236–251

[11] Sharp, R., Mycroft, A.: The flash compiler: Efficient circuits from functional specifications. Technical Report tr.2000.3, AT&T Research (2000)

[12] Mycroft, A., Sharp, R.: A statically allocated parallel functional language. In: ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming, Springer-Verlag (2000) 37–48

[13] Wadler, P.: Monads for functional programming. In Broy, M., ed.: Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School, Springer-Verlag (1993)

[14] Wadler, P.: The essence of functional programming. In: Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albequerque, New Mexico (1992) 1–14

[15] Wadler, P.L.: Comprehending monads. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice, New York, NY, ACM (1990) 61–78

[16] Espinosa, D.: Semantic Lego. PhD thesis, Columbia University (1995)

[17] Duponcheel, L.: Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters. (1995)

[18] Hutton, G.: Fold and unfold for program semantics. In: Proceedings 3rd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'98, Baltimore, MD, USA, 26–29 Sept. 1998. Volume 34. ACM Press, New York (1998) 280–288

[19] Labra Gayo, J.E., L., J.M.C., D., M.C.L., del Rio, A.C.: Reusable monadic semantics of object oriented programming languages (2002)

[20] Labra Gayo, J.E., Luengo Díez, M.C., Cueva Lovelle, J.M., Cernuda del Río, A.: LPS: A language prototyping system using modular monadic semantics. In van der Brand, M., Parigot, D., eds.: Proceedings 1st Workshop on Language Descriptions, Tools and Applications, LDTA'01, Genova, Italy, 7 Apr 2001. Volume 44(2). Elsevier, Amsterdam (2001)

[21] Labra Gayo, J.E., Luengo Díez, M.C., Cueva Lovelle, J.M., Cernuda del Río, A.: Reusable monadic semantics of logic programs with arithmetic predicates. In: Proceedings 2001 APPIA-GULP-PRODE Joint Conf. on Declarative Programming, AGP'01, Évora, Portugal, 26–28 Sept. 2001. Dept. of Informatics, Univ. of Évora (2001) 31–45

[22] Lammel, R., Visser, J., Kort, J.: Dealing with large bananas. In Jeuring, J., ed.: Workshop on Generic Programming. (2000)

[23] Kimmell, G., Komp, E., Alexander, P.: Building compilers by combining algebras. In: ECBS, IEEE Computer Society (2005) 331–338

[24] Goodman, H.S.: Animating Z specifications in Haskell using a monad. Technical report CSR-93-10, School of Computer Science, University of Birmingham, Birmingham, England, B15 2TT (93)

[25] Abdallah, A.E., Barros, A., Barros, J.B., Bowen, J.P.: Deriving correct prototypes from formal Z specifications. Technical Report SBU-CISM-00-27, South Bank University, SCISM, London, UK (2000)

[26] Lämmel, R.: Adding superimposition to a language semantics (extended abstract). In: Foundations of Aspect Oriented Languages. (2003)

[27] Mudge, T.: Power: A first-class architectural design constraint. IEEE Computer (2001) pp 52-58.

[28] Kim, N.S., Austin, T., Mudge, T., Grunwald, D.: Challenges for Architectural Level Power Modeling. In: Power Aware Computing. Kluwer (2001)

[29] Filinski, A.: Representing monads. In: Conf. Record 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'94, Portland, OR, USA, 17–21 Jan. 1994. ACM Press, New York (1994) 446–457

[30] Filinski, A.: Representing layered monads. In: POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (1999) 175–188