Modular Monadic Semantics

Sheng Liang Paul Hudak

Department of Computer Science
Yale University
P.O. Box 208285
New Haven, CT 06520-8285, USA
{liang,hudak}@cs.yale.edu

Abstract

Modular monadic semantics is a high-level and modular form of denotational semantics. It is capable of capturing individual programming language features as small building blocks which can be combined to form a programming language of arbitrary complexity. Interactions between features are isolated in such a way that the building blocks are invariant. This paper explores the theory and application of modular monadic semantics, including the building blocks for individual programming language features, equational reasoning with laws and axioms, modular proofs, program transformation, modular interpreters, and semantics-directed compilation. We demonstrate that modular monadic semantics makes programming languages easier to specify, reason about, and implement than the alternative of using conventional denotational semantics.

Our contributions include: (a) the design of a fully modular interpreter based on monad transformers, including important features missing from several earlier efforts, (b) a method to lift monad operations through monad transformers, including difficult cases not achieved in earlier work, (c) a study of the semantic implications of the order of monad transformer composition, (d) a formal theory of modular monadic semantics that justifies our choice of liftings based on a notion of naturality, and (e) an implementation of our interpreter in Gofer, whose constructor classes provide just the added power over Haskell type classes to allow precise and convenient expression of our ideas.

A note to reviewers: this paper is rather long. Short of resorting to "Part I / Part II", the one way we see to shorten it would be to remove Section 4 and its Appendix B, which would amount to eliminating contribution (e) above. This would shorten the paper by about 12 pages.

1 Introduction

1.1 Overview

Denotational semantics (Stoy, 1977) is among the most important developments in programming language theory. It gives a precise mathematical description of programming languages, useful in designing and implementing languages as well as reasoning about programs. For example, advances in denotational semantics have led to clarifications of features, to more consistent programming language design, and to new programming constructs.

It has long been recognized, however, that traditional denotational semantics lacks

modularity and extensibility (Mosses, 1984) (Lee, 1989). This is regarded as a major obstacle in applying denotational semantics to realistic programming languages.

In this paper, we take advantage of a new development in programming language theory—a monadic approach (Moggi, 1990) to structured denotational semantics. The resulting *modular monadic semantics* achieves a high level of modularity and extensibility. It is able to capture individual programming language features in reusable building blocks, and to specify programming languages by composing the necessary features.

Because modular monadic semantics is no more than a structured denotational semantics, standard equational reasoning methods still apply. In addition, we show that modular monadic semantics further facilitates reasoning by allowing us to specify axioms of programming language features and to construct reusable modular proofs.

Modular monadic semantics can be implemented using modern programming languages such as Haskell (Hudak *et al.*, 1992), ML (Milner *et al.*, 1990), or Scheme (Clinger & Rees, 1991). The result is a modular interpreter. We have discovered, however, that the relatively new idea of *constructor classes* in Gofer (and Haskell 1.3) are particularly suitable for representing some rather complex typing relationships in modular interpreters, and thus we choose Gofer for the interpreter described in Section 4. Our work is also directly applicable to semantics-directed compiler construction, and we present a compilation method based on monadic semantics and monadic program transformations.

Before introducing modular monadic semantics, in the next section we give an example to demonstrate the lack of modularity in traditional denotational semantics. The presentation follows the traditional denotational semantics style, augmented with a types declaration syntax similar to that of Haskell or ML. We assume the reader has basic knowledge of denotational semantics and functional programming.

1.2 The Lack of Modularity in Denotational Semantics

Let us first look at the denotational semantics of a simple arithmetic language:

$$
\begin{array}{lcl}
E & : & \mathit{Term} \to \mathit{Value} \\
E[\![n]\!] & = & n \\
E[\![e_1 + e_2]\!] & = & E[\![e_1]\!] + E[\![e_2]\!]
\end{array}
$$

Denotational semantics maps *terms* in the source language into *values* in the meta language. The source language terms are enclosed in "$[\![\ ]\!]$". The n and $+$ symbols on the right hand side correspond to the meta language concepts of a *number* and the *add* arithmetic operation.

An important measure of modularity is how a semantic description can be extended to incorporate new programming language features. For example, if we extend the source language with variables and functions, we need to introduce an environment—a mapping from variable names to values:

$$
\begin{array}{lcl}
E & : & \mathit{Term} \to \mathit{Env} \to \mathit{Value} \\
E[\![n]\!] & = & \lambda\rho.n \\
E[\![e_1 + e_2]\!] & = & \lambda\rho.E[\![e_1]\!]\rho + E[\![e_2]\!]\rho \\
E[\![v]\!] & = & \lambda\rho.\rho[\![v]\!]
\end{array}
$$

Note that even though numbers are independent of the environment, we must change the semantics of numbers to accommodate the newly introduced environment argument. Similarly, the environment argument must be passed recursively to the subexpressions of $e_1 + e_2$, even though the arithmetic operation itself is independent of the environment.

If we further add continuations to our semantics (for supporting, for example, the sequencing operator ";"), we must change the semantics of numbers once again:

$$
\begin{array}{lcl}
E & : & \mathit{Term} \to \mathit{Env} \to (\mathit{Value} \to \mathit{Ans}) \to \mathit{Ans} \\
E[\![n]\!] & = & \lambda\rho.\lambda k.kn \\
E[\![e_1 + e_2]\!] & = & \lambda\rho.\lambda k.E[\![e_1]\!]\rho(\lambda i.E[\![e_2]\!]\rho(\lambda j.k(i+j))) \\
E[\![e_1; e_2]\!] & = & \lambda\rho.\lambda k.E[\![e_1]\!]\rho(\lambda x.E[\![e_2]\!]\rho k)
\end{array}
$$

In summary, we must make global changes to the traditional denotational semantics in order to add new features into the source language. This lack of modularity of denotational semantics has long been recognized (Mosses, 1984) (Lee, 1989), and is regarded by many as the most significant obstacle in applying denotational semantics to realistic programming languages.

1.3 Monads to the Rescue

Consider now a type constructor M and two functions:

$$
\begin{array}{lcl}
\mathit{return} & : & a \to M\ a \\
\mathit{bind} & : & M\ a \to (a \to M\ b) \to M\ b
\end{array}
$$

The intuitive meanings of these operations are as follows:

- $M\ a$ is a *computation* returning a value of type a.
- $\mathit{bind}\ c_1\ (\lambda v.c_2)$ is a computation that first computes c_1, binds the result to v, and then computes c_2.
- $\mathit{return}\ v$ is a trivial computation that simply returns v as result.

With these operations we can rewrite the semantics for arithmetic expressions as follows:

$$
\begin{array}{lcl}
E & : & \mathit{Term} \to M\ \mathit{Value} \\
E[\![n]\!] & = & \mathit{return}\ n \\
E[\![e_1 + e_2]\!] & = & \mathit{bind}\ (E[\![e_1]\!]) \\
 & & \quad (\lambda i.\ \mathit{bind}\ (E[\![e_2]\!]) \\
 & & \quad\quad (\lambda j.\ \mathit{return}\ (i+j)))
\end{array}
$$

Note now that the semantic function E maps terms to *computations* (of type $M\ \mathit{Value}$). The above equations can be read: "the semantics of $E[\![n]\!]$ is a trivial computation that returns n as result; the semantics of $E[\![e_1 + e_2]\!]$ is a computation that computes $E[\![e_1]\!]$, binds the result to i, computes $E[\![e_2]\!]$, binds the result to j, and finally returns $i + j$."

We call this a *parameterized semantics* because, depending on how we instantiate M, *return* and *bind*, we get different concrete semantics. For example, Figure 1 shows how the arithmetic semantics can be instantiated to the trivial and environment-based semantics described earlier. To give meaning to variables in the context of the environment-based semantics, we simply add the equation:

$$
\begin{array}{lcl}
E[\![v]\!] & = & \mathit{bind}\ (\mathit{rdEnv}) \\
 & & \quad (\lambda\rho.\ \mathit{return}(\rho[\![v]\!]))
\end{array}
$$

where *rdEnv* (defined in a later section) is a computation that reifies the environment. The key point here is that the previous equations did not need to be altered. In a similar way, we show later that appropriate definitions of M, *return* and *bind* can yield the continuation-based semantics discussed earlier, as well as several other important semantics to support other programming language features.

The type constructor M, together with the two functions *return* and *bind*, are called a *monad*, and a parameterized semantics using monads is called a *monadic semantics*. A monadic semantics can be instantiated using different *underlying monads*. In general, to add a new feature to a monadic semantics, we only need to add a semantic description of the new feature and change the underlying monad, but no changes are required of the semantic descriptions of the existing features.