Monads

Mark Hills
mhills@cs.uiuc.edu

Department of Computer Science
University of Illinois at Urbana-Champaign

6 August 2009
Overview

- Introduction to Monads
- Research Directions
- Comparison with K
In *An Abstract View of Programming Languages* (Moggi, 1989), Moggi refers to an upcoming book chapter by Peter Mosses detailing both successes and failures of denotational semantics, the failures include:

- “the denotations of simple expressions, e.g. integer expressions, might have to be changed when the programming language is extended”;
- “Denotational Semantics is feasible for toy programming languages, but does not scale up easily to real programming languages”
- “it is not feasible to re-use parts of the description of one programming language in another”

The core problem is identified as a lack of **modularity**.
The Intuition

- Modularity problems caused by needing to “thread” configuration through semantics – change the configuration, change the semantic functions
The Intuition

- Modularity problems caused by needing to “thread” configuration through semantics – change the configuration, change the semantic functions
- So, throw all the configuration into a box, pass the box around instead – the configuration essentially becomes an ADT
The Intuition

- Modularity problems caused by needing to “thread” configuration through semantics – change the configuration, change the semantic functions
- So, throw all the configuration into a box, pass the box around instead – the configuration essentially becomes an ADT
- Plus, provide ways to combine boxes, allowing changes without requiring changes to the underlying functions
Notions of Computation

- Denotational semantics maps syntactic entities to their denotations
- Denotations based on domains (from domain theory), formed using domain formation operations
- Monads modifies this: programs now map values to *computations*
Categorical Understanding (High-Level)

- Traditional: programs viewed as morphisms of a category of types
- Monadic:
  - Start with a category $C$, made up of objects $A$ (representing sets of type $\tau$) and $T A$ (representing computations of type $\tau$)
  - Identify programs from type $A$ to $B$ with morphisms from $A$ to $T B$ in $C$
  - Impose rules on values and computations to ensure that programs are morphisms in the desired category
Monad Laws

Laws formulated in terms of \(\textit{unit}\) and either \(\textit{bind}\) or \(\textit{map}\) an \(\textit{join}\); most older papers seemed to use \(\textit{map}\) and \(\textit{join}\), most newer (plus Haskell) use \(\textit{bind}\)

\[
\text{unit} :: a \rightarrow M a \tag{1}
\]

\[
\text{map} :: (a \rightarrow b) \rightarrow (M a \rightarrow M b) \tag{2}
\]

\[
\text{join} :: M (M a) \rightarrow M a \tag{3}
\]

\[
\text{bind} :: M a \rightarrow (a \rightarrow M b) \rightarrow M b \tag{4}
\]
Monadic Interpreters, Part 1

- *Comprehending Monads* (Wadler, 1990), and *The Essence of Functional Programming* (Wadler, 1992) both introduced monads as a method of structuring functional programs, especially those focused on interpreting languages.

- *Combining Monads* (King and Wadler, 1992) and *Composing Monads* (Jones and Duponcheel, 1993) further investigated ways to combine monads, trying to determine under which conditions this was well defined (and fairly automatic).
Building Interpreters by Composing Monads (Steele, 1994) continued this, showing methods to compose monads using premonads, but encountering trouble with typing.

Monad Transformers and Modular Interpreters (Liang, Hudak, and Jones, 1995) and Modular Denotational Semantics for Compiler Construction (Liang and Hudak, 1996) continued this focus, also investigating ways to combine monads (using Moggi’s idea of monad constructors or monad transformers).
Denotational Semantics

- *Semantic Lego* (Espinosa, 1995) provided a Scheme-based environment for generating language definitions based on monadic semantics, using a technique called *stratification* to combine monads.

- *A Syntactic Approach to Modularity in Denotational Semantics* (Cenciarelli and Moggi, 1993), *Composing Monads Using Coproducts* (Lüth and Ghani, 2002), and probably others, have continued with a focus on notions of modularity and combining monads.

- *Modularity in Denotational Semantics* (Power, 1997), and probably others, have maintained a focus on modularity of denotational definitions, but not necessarily using the same techniques.
Overview: Benefits

- Definitions maintain good aspects of denotational semantics; functional style, compositionality, clear mathematical meaning (we have third, but not first, and we don’t worry about second)
- Works with established proof techniques
- Close relationship to functional languages
Benefit: Relationship to Denotational Semantics

- Monadic definitions still use underlying denotational style of definition, “de-cluttered” by removing many references to state; provides familiar environment for semanticists, access to already-developed semantic techniques
- Like other denotational definitions, monadic definitions are compositional, something not enforced by K
- Denotational definitions have a well-understood mathematical meaning (so does K, at least when viewed in light of rewriting logic, so this isn’t a benefit over K, just a benefit)
Monadic definitions still use underlying denotational style of definition, “de-cluttered” by removing many references to state; provides familiar environment for semanticists, access to already-developed semantic techniques.

Like other denotational definitions, monadic definitions are compositional, something not enforced by K.

Denotational definitions have a well-understood mathematical meaning (so does K, at least when viewed in light of rewriting logic, so this isn’t a benefit over K, just a benefit).

Of these benefits, compositionality is probably the strongest contrast with K.
Many proof techniques developed for denotational definitions over the years (for a recent example, see LOOP, which provided a denotational semantics for JavaCard).

Monadic definitions are just “well-organized” denotational definitions, can leverage existing proof techniques plus provide an organizational structure for modular proofs.
Benefit: Proof Techniques

- Many proof techniques developed for denotational definitions over the years (for a recent example, see LOOP, which provided a denotational semantics for JavaCard)
- Monadic definitions are just “well-organized” denotational definitions, can leverage existing proof techniques plus provide an organizational structure for modular proofs

This is probably the area where we are weakest in comparison.
Benefit: Functional Languages

- Close relationship with functional languages provides a natural platform for implementing ideas
- Also has had a strong influence on functional programming, providing a compelling organizational principle for otherwise pure functional code
- Relationship with functional languages ensures good tool support for working with definitions

Not the strongest benefit, but providing a familiar environment, with good tool support, definitely increases acceptance.
Overview: Drawbacks

- Underlying theory quite complex, can make it challenging for those outside of semantics community (although integration with functional languages helps)
- Along with maintaining benefits of denotational definitions, maintains some shortcomings as well, such as limitations in concurrency semantics
- Modularity based on composing monads, but this has proven tricky in practice
- Monad transformers not modular, order of application of transformers can impact final semantics
Drawback: Complexity and Limitations (Part 1)

- Underlying theory of monads quite complex, uses some fairly advanced mathematical concepts
- Brings into question how effectively monadic definitions can be used by “regular” computer scientists
- Term rewriting, equational logic, and rewriting logic more accessible (in my opinion)
Rewriting logic provides a “cleaner” definition of nondeterminism; monads inherit difficulties of denotational semantics, most related work does not even address this (or uses techniques like resumptions)

Note: relationship with functional languages helps here for some parts, but most complex part of learning Haskell is: monads, plus this creates another “layer” – language semantics based on Haskell semantics, but do we know they are correct?
K definitions do not need to worry about composition, context transformers work to transform semantics “at the end”, once a final configuration has been determined, in one shot.

Monadic definitions do need to worry about composition; complex monads constructed from simpler monads, but monads do not normally compose into monads, requiring other techniques (monad transformers).

In some cases (continuations), there is no general technique to compose monads, requiring ad-hoc techniques to be used instead; this should only happen in K in very odd situations, like using rules that use the same K cell but use it for different purposes.
Drawback: Monad Transformers not Modular

- Monad transformers provide a way to combine monads by lifting operations in one through another.
- Since the order of composition matters, it is important to know what monads are already being used when adding a new one through a transformer.
- It is also important to know the orders, since the ordering is a "global" property of the definition – quadratic number of possible orderings.
- K does not need individual state transformers for each feature, plus the requirement to only transform cell nesting should allow for context transformers to be modular if need be.