Relative Monads in Call-by-push-value as an Abstraction of Stack-based Effects

(Extended Abstract)

Max S. New
Computer Science and Engineering
University of Michigan
Ann Arbor, Michigan, USA
maxsnew@umich.edu

Abstract

We propose the use of relative monads in a Call-by-push-value calculus as a useful abstraction for stack-based effect implementations.

1 Introduction

Since Moggi's seminal work [Moggi 1991], monads have become a wildly successful tool in two main areas. First, as Moggi originally showed, denotational models of functional languages with effects are naturally modeled using monads. Second, monads have become an indispensable programming abstraction for effects in functional languages, most notably in Haskell [Peyton Jones and Wadler 1993].

While monads have been very successful in modeling high-level, especially user-defined, effects, they have not seen as much use in the practice of low-level effect implementation in compilers, which involve invariants on the use of registers and the stack that are not naturally modeled in the usual semantics of an effect as a strong monad. The most successful applications rely on converting the monads to continuation-passing style and applying the usual CPS compilation techniques.

We propose a simple model for stack-based effect implementations as (strong) relative monads in a call-by-push-value calculus. These are sufficiently close conceptually to ordinary strong monads that much of the intuitions around monads still apply, but incorporate a distinction between the linearly used runtime stack first class values. The foundation for this insight is given by the call-by-push-value calculus, which itself is based on a refinement of Moggi's analysis of effects in terms of monads.

2 Stack structure via Computation Types

First, we introduce our metalanguage: Levy's Call-by-push-value (CBPV) calculus extended with recursive types and polymorphism[Levy 2001]. CBPV is based on a fundamental distinction between *value types* which describe sets of first-class values and *computation types* which describe the possible behaviors of computations that interact with the environment. The type structure of this calculus is given in Figure 1. The value types are written as A and should mostly be familiar: nullary and binary tuples along with nullary and binary sum types. The last, $U\underline{B}$ is the type of *thUnks* of computations of type $\underline{B}$. Computation types $\underline{B}$ are more interesting. In order to describe behaviors that interact with the environment, computation types must also describe the *expectations* on the structure of the external environment. Most importantly for this work, the computation types can be interpreted as defining expectations on the structure of a runtime stack. For instance, the CBPV function type $A \to \underline{B}$ describes computations that pop an A value off of the stack and proceed to behave as $\underline{B}$. In terms of the stack structure, this requires the stack to consist of an A value pushed onto a $\underline{B}$ stack. The lazy product $\underline{B}_1 \,\&\, \underline{B}_2$ describes a computation that, depending on the external environment, either behaves as a $\underline{B}_1$ or as a $\underline{B}_2$. This requires the stack to consist of a bit to indicate which behavior is desired, paired with an appropriate stack for either $\underline{B}_1$ or $\underline{B}_2$.[1] We also include a type of coinductively defined computations $\nu\underline{Y}.\underline{B}$ which have inductive expectations on the stack, and polymorphic computations $\forall\underline{Y}.\underline{B}$ that have a similar interpretation to function types. Finally, we have the $\underline{F}A$ type which is the type of programs that perform effects and (possibly) return values of type A. In terms of the stack, this requires that the stack provide a continuation for A values as well as whatever other features are needed to allow the allowed effects of the language such as exceptions or state.

Levy's CBPV calculus can be viewed as a refinement of Moggi's monadic metalanguage. Moggi's metalanguage has a single sort of types with a type TA representing the "effectful computations" that return A values. This intuitively is very similar to the $\underline{F}$ type, but the crucial difference is that terms of type TA are first class data that can be passed around. In

Author's address: Max S. New, Computer Science and Engineering, University of Michigan, Ann Arbor, Michigan, USA, maxsnew@umich.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

[1] The duality between computation types and stacks is an instance of categorical/linear logic duality, see Møgelberg and Staton [2014] for more details

$$\begin{array}{rrcl} \text{Value types} & A & ::= & U\underline{B} \mid 1 \mid A \times A \mid 0 \mid A + A \\ \text{Computation types} & \underline{B} & ::= & \underline{Y} \mid \underline{F}A \mid A \to \underline{B} \mid \nu \underline{Y}.\underline{B} \\ & & & \mid \forall \underline{Y}.\underline{B} \mid \top \mid \underline{B} \& \underline{B} \end{array}$$

Figure 1. Extended CBPV Types

CBPV, these two aspects of the monad T are decomposed into a type of effectful behaviors $\underline{F}$ and a type of first class closures U. These constructors can be combined to construct a strong monad on the category of value types $U\underline{F}A$. In category-theoretic terms, the constructors $\underline{F}$ and U are a strong adjunction and every adjunction induces a monad[2].

3 Relative Monads for Stack-based Effects

CBPV generalizes Moggi's metalanguage to give a calculus for programming with an *abstract* effect, but we can also use CBPV as a metalanguage for describing concrete effects, just as in Haskell, users can describe concrete effects using the `Monad` typeclass, which abstracts over the structure of the type T in Moggi's metalanguage. To formalize the structure of stack-based effect implementations in CBPV, we should instead abstract over the structure of the type $\underline{F}$ in CBPV. A natural way to do this is to axiomatize the structure of a monad *relative to* $\underline{F}$, using the concept of relative monad developed in Altenkirch et al. [2010]. Relative monads are not endofunctors, in this case a monad relative to $\underline{F}$ is given by a type constructor $\underline{\mathrm{Eff}}A$ that takes value types to computation types, like $\underline{F}$, along with some term constructors analogous to Haskell's `monad` typeclass. This can be described in several ways, the simplest of which can be described by the following pseudo-code for a CBPV typeclass[3]:

```
class RMonad (Eff : v -> c) where
  return : a -> Eff a
  bind   : U(Eff a) -> U(a -> Eff a') -> Eff a'
```

subject to monad laws analogous to the ordinary laws, as well as a constraint that `bind` is linear in its input in the sense of Munch-Maccagnoni [2014].

The intuition for this type is slightly different than a typical monad. We think of the type $\underline{\mathrm{Eff}}A$ as the type of computations that return A values, and therefore also the *expectations* on the stack for such computations. Then `return` says we can return an A value to an $\underline{\mathrm{Eff}}$ stack, and `bind` says that we can *extend* a stack for $\underline{\mathrm{Eff}}A'$ computations with a continuation $U(A \to \underline{\mathrm{Eff}}A')$ to construct a stack for $\underline{\mathrm{Eff}}A$ computations to interact with.

3.1 Example: Three Exception Monads

We end with a concrete example of how CBPV relative monads can be used to describe stack-based implementations of effects more faithfully than ordinary monads. In Figure 2 we present three relative exception monads. The first is simply the ordinary exception monad implementation using sums extended to be a relative monad. This representation is known to introduce unnecessary runtime overhead because the continuation for the exception must inspect a sum value and branch based on its structure.

$$\begin{aligned} \mathrm{Exn}_0 EA &= \underline{F}(E + A) \\ \mathrm{Exn}_1 EA &= \forall \underline{Y}.U(A \to \underline{Y}) \to U(E \to \underline{Y}) \to \underline{Y} \\ \mathrm{Exn}_2 EA &= (F(A + E)) \,\&\, (\forall X.U(A \to \mathrm{Exn}_2 EX) \to \mathrm{Exn}_2 EX) \end{aligned}$$

Figure 2. Three Relative Exception Monads

The second representation is a relative variant of the "double-barrel continuation" monad [Thielecke 2001]. This representation is a variant of CPS conversion where both "success" and "fail" continuations are provided. This eliminates the indirection overhead of the first representation because the monadic computation directly jumps to the appropriate continuation rather than constructing a value of a sum type. This type is simply a Church-encoding of the first, and it can be shown using dinaturality or parametricity to be equivalent to the first[Mogelberg and Simpson 2007].

The third representation is a stack-walking implementations of exceptions[4]. Stack-walking implementations optimize for the scenario where exceptions are rare. The double-barrel representation is not ideal for this scenario since each bind requires pushing an additional continuation handler onto the stack. In the stack-walking implementation, the computation is a &, offering a choice to the environment: either the environment provides a continuation for an ordinary exception monad, or it provides a continuation for the A values only in addition to the rest of the stack. This means that when raising an exception, the stack might have an arbitrary number of value continuations before providing an exception handler. The code for raising an exception must then be a recursive function that must walk up the stack until it finds an exception handler. Note that the type Exn_2 is slightly too large, in order to be equivalent to the previous two, we should restrict to a subset of behaviors that return the same A value no matter which choice is provided by the context.

4 Future Work

Relative version of other typical monads, such as reader, writer, state, continuations and free monads can all be formulated in CBPV. An optimizing implementation of CBPV would allow us to implement these and evaluate the effect on performance.

[2] as well as a comonad

[3] alternative formulations make the role of the $\underline{F}$ type much clearer

[4] Note that this definition is a coinductive definition that needs higher order ν

While CBPV allows us to model stack manipulation using computation types, it also suffers from being "too high level" in a few ways. For instance, the double-barreled continuation monad uses two closures, but in a real implementation, the captured variables of these two continuations can be shared and stored on the stack. Then instead of two closures, we could have simply two code pointers. This would require a lower-level version of CBPV that has a distinction between closures and more low level code pointers that cannot capture free variables. Additionally, runtime systems often reserve certain registers for their effect implementations. CBPV only models the structure of the stack explicitly, but possibly an extension can model global registers as well, for instance implementing a form of relative state monad where the value is updated in place in a register.

References

Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. 2010. Monads Need Not Be Endofunctors. In *Foundations of Software Science and Computational Structures*. 297–311.

Paul Blain Levy. 2001. *Call-by-Push-Value*. Ph. D. Dissertation. Queen Mary, University of London, London, UK.

Rasmus Ejlers Mogelberg and Alex Simpson. 2007. Relational Parametricity for Computational Effects. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. 346–355. https://doi.org/10.1109/LICS.2007.40

Eugenio Moggi. 1991. Notions of computation and monads. *Inform. And Computation* 93, 1 (1991).

Guillaume Munch-Maccagnoni. 2014. Models of a Non-associative Composition. In *Foundations of Software Science and Computation Structures*, Anca Muscholl (Ed.). 396–410.

Rasmus Ejlers Møgelberg and Sam Staton. 2014. Linear usage of state. *Logical Methods in Computer Science* Volume 10, Issue 1 (March 2014). https://doi.org/10.2168/LMCS-10(1:17)2014

Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, 71–84. https://doi.org/10.1145/158511.158524

Hayo Thielecke. 2001. Comparing Control Constructs by Double-barrelled CPS Transforms. *Electronic Notes in Theoretical Computer Science* 45 (2001), 433–447. https://doi.org/10.1016/S1571-0661(04)80974-5 MFPS 2001,Seventeenth Conference on the Mathematical Foundations of Programming Semantics.