Abstraction and Performance from Explicit Monadic Reflection

Jonathan Sobel
SAS Institute
jsobel@acm.org

Erik Hilsdale
Google Inc.
eh@acm.org

R. Kent Dybvig
Daniel P. Friedman *
Indiana University
{dyb,dfried}@cs.indiana.edu

Abstract

Most of the existing literature about monadic programming focuses on theory but does not address issues of software engineering. Using monadic parsing as a running example, we demonstrate monadic programs written in a typical style, recognize how they violate abstraction boundaries, and recover clean abstraction crossings through monadic reflection. Once monadic reflection is made explicit, it is possible to construct a grammar for monadic programming that is independent of domain-specific operations. This grammar, in turn, enables the redefinition of the monadic operators as macros that eliminate at expansion time the overhead imposed by functional representations. The results are very efficient monadic programs; for parsing, the output code is competitive with good hand-crafted parsers.

1. Introduction

The use of monads to model effect-laden computation has become commonplace. This work aims to show that a fuller appreciation of the theory of monads can improve the correctness and efficiency of such implementations. We explore this through a single application domain: parsing. First, we approach parsing from the functional perspective. Next, we observe some of the shortcomings of overly simplistic monadic programming and observe what happens when we change our language to fit the theory more closely. We then explore the efficiency improvements such a foundation allows us. Finally, we point toward how the parsing example we use may be generalized.

Most of the presentation in the following section is not new. Using monads for parsing has been discussed in detail by Wadler [18], Hutton [7] and Meijer [8, 9], and Bird [1]. In a change from these presentations, however, the programs in this paper are written in the strict language Scheme [10] and include uses of Scheme's syntactic-extension mechanism (macros). We paraphrase the material from these other texts in order to familiarize the reader with our terminology and notation.

One might reasonably ask why, when exploring a topic that involves very typeful monads and their associated operators, would the presentation use the dynamically-typed language Scheme? The answer is two-fold. First, the goals of this work are more in the realm of software engineering than theory. The monads and types are useful vehicles for understanding the programs, but the true target is easy-to-write, easy-to-maintain, efficient software. Choosing Scheme should not *prevent* the use of monads for structuring programs. Second, this presentation relies heavily on syntactic abstraction as a means of turning programming patterns into language extensions, which can then be reimplemented as more efficient patterns. Such an approach is sadly impossible in any common statically-typed language.

In Section 3 we draw an analogy between monads and abstract data types. Such an analogy is not new; the example of the simple state monad with "get" and "set" operations is often presented as an abstract data type. The problem is that in larger, more realistic examples—such as functional parsing—the number of operations that requires access to the monad's underlying representation is much larger. When seen in this light, it becomes clear that a significant portion of the typical monadic-style program is treated as if it falls *inside* the abstraction boundary of the abstract data type. To complicate matters, it is very difficult for the provider of the monad data type to guess every operation that real client code might need. A review of the definition of monads leads us to monadic reflection, which provides the right tools to draw a new boundary between the very few core monad operations and the many operations that need to be partially aware of the monad's underlying representation. We rewrite portions of the code from Section 2 in a cleaner style using monadic reflection. The reflection operators, together with the standard monadic programming operators, provide enough expressiveness for us

* This work was supported in part by the National Science Foundation under grant CCR-9633109.

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.

to construct a grammar for the sublanguage of monadic programs. This grammar supports three-layer monadic programming: the monad definition itself, representation-aware operators, and representation-independent client code. The three-layer model stands in contrast to the typical two-layer model where everything other than the client code is treated as part of the core monad definition.

Once we have a specification of monadic programs, we are in a good position to optimize them. This we do by changing the definitions of the monadic operators in Section 4 while leaving their interfaces intact. All unnecessary closure creation is eliminated, and the work of threading store/token-stream values through the computation is handled entirely at expansion time in the new definitions. Programs that conform to our monadic-programming grammar need not be rewritten at all to benefit from the optimizations. Furthermore, all the optimizations are handled at the source level by user-defined macros, not by a new compiler pass. The approach described here is relevant for any composition of store-like monads, possibly composed with a lifting or error monad.

2. Parsing

Parsers are often described as functions from token streams to abstract syntax trees:

$$\mathit{Parser} = \mathit{Tokens} \to \mathit{Tree}$$

This characterization does not account for parsers modifying the token stream. That is, by the time the parser produces a tree, the token stream no longer has its original contents. Thus, the type needs to be revised:

$$\mathit{Parser} = \mathit{Tokens} \to \mathit{Tree} \times \mathit{Tokens}$$

It could be the case, though, that the parser fails to construct a tree (for example, if the input is malformed). To handle this possibility, we lift the Tree type to $\mathit{Tree} + \mathit{ErrMsg}$:

$$\mathit{Parser} = \mathit{Tokens} \to (\mathit{Tree} + \mathit{ErrMsg}) \times \mathit{Tokens}$$

(This compact type will continue to appear in the remainder of this article, but for efficiency the programs actually use

$$\begin{aligned}&\mathit{Parser} = \\ &\quad \mathit{Tokens} \to (\mathit{Tree} \times \mathit{Tokens}) + (\mathit{ErrMsg} \times \mathit{Tokens})\end{aligned}$$

which is isomorphic to the prior type by the distributive property.)

The preceding paragraph follows the standard sequence of types and justifications to arrive at a desirable type for parsers,[1] but we find that the effect is to direct one's attention the wrong way. We want primarily to think about the parser's results. Parsers, however they operate, produce trees. Yet most of the type we specified for parsers is not about trees; it's about the wiring that gives us the trees. Instead, let's just say that *parsing* (not *parsers*) is one way to describe tree-producing computation. Henceforth, we shall refer to tree-producing computations (or just *tree producers*) instead of parsers.

Trying to talk about computations presents us with a problem: how do we manipulate computations in programs? We need something to act as a "representation of a tree producer." Exactly how we represent these computations depends on what aspects we want to model. Above, in the context of traditional parsing technology, we arrived at functions of a certain shape as our representations. Specifically, our representation modeled the threading of a token stream through the computation [16], as well as the possibility of failure. We call this a *threaded functional representation* of a tree producer. Let's express this abstraction in the type constructor $\mathit{Producer}$:

$$\mathit{Producer}(\alpha) = \mathit{Tokens} \to (\alpha + \mathit{ErrMsg}) \times \mathit{Tokens}$$

Thus, $\mathit{Producer}(\mathit{Tree})$ is our representation for computations that produce trees.

The sum type can be represented in many ways in Scheme. For injecting values into the left and right sides of the sum, we use the operators `inl` and `inr`, respectively. These operators are polymorphic over the number of injected values, so `(inl x y z)` is acceptable usage. For dispatching on the two sum cases, we use the `sum-case` form.

⟨*example of* `sum-case`⟩≡
```
(sum-case (inl 5 2)
  ((x y) (+ x y))
  ((a b) (- a b)))
```

The value of this expression is 7. A portable implementation of `inl`, `inr`, and `sum-case` appears in the appendix. Additional options for representing sums and a discussion of their performance implications appears in Section 4.3.

It would be inconvenient to write parsers if we had to explicitly manage values of the $\mathit{Producer}$ types. Monads provide just the right additional structure for manipulating these values, so that programs have a consistent style, and so that the details of the $\mathit{Producer}$ types are abstracted away [13, 14, 19].

To make this claim more concrete, let us construct a little program in Scheme for parsing natural numbers (non-negative integers). We begin with a version written *without* the benefit of monadic operators. Even those readers who are already quite familiar with monads may find it interesting to follow the derivation of monadic structure as a kind of "pattern-mining" via syntactic abstraction.

[1] Allowing a failed parse to return a new token stream is not really standard in the literature. Why do we allow it here? Because implementations based on real imperative input streams often modify the stream even on a failed parse. In fact, such behavior is often desirable in a robust parser, to eliminate nonsense tokens from the input and continue to make progress.

2.1 Parsing Natural Numbers

A program that reads the digits in its input and parses numbers would be more typically described as scanning, not parsing, but if we take individual characters as our tokens, the distinction becomes largely moot. Here is a grammar for natural numbers:

$$\begin{array}{lcl} \langle\text{natural}\rangle & \rightarrow & \langle\text{digit}\rangle\ \langle\text{more digits}\rangle \\ \langle\text{more digits}\rangle & \rightarrow & \langle\text{digit}\rangle\ \langle\text{more digits}\rangle \\ & \mid & \langle\text{empty}\rangle \end{array}$$

The entry point for our program is the procedure `natural`,[2] which is intended to instantiate an integer-producing computation:

⟨*long version of* `natural`⟩≡
```
(define natural
  (lambda ()
    ⟨integer producer for natural⟩))
```

Using our representation scheme for computations, this means that `natural` should return a value of type

$$\mathit{Producer}(\mathit{Int}) = \mathit{Tokens} \rightarrow (\mathit{Int} + \mathit{ErrMsg}) \times \mathit{Tokens}$$

Now let's assume the existence of a nullary procedure `digit`, which returns a character producer that gets a numeric character from the token stream. It fails (i.e., returns an error message) if the next available character is not a digit or if no characters are available. Since a natural number begins with at least one digit, we get:

⟨*integer producer for* `natural`⟩≡
```
(lambda (ts1)
  (sum-case ((digit) ts1)
    ((d ts2) (⟨integer producer, given first digit⟩ ts2))
    ((msg ts2) (inr msg ts2))))
```

The values returned by `digit` are of the sum type, so we must use `sum-case` to determine whether `digit` failed or not. If so, then `natural` itself must also fail, returning the bottom value and the new tokens `ts2`. (Failures get to eat tokens, too.) The rest of the number comes from `more-digits`—to be defined shortly—which instantiates a list-producing computation, giving us a list of all the digits (numeric characters) it can extract from the front of the token stream. The portion that reads the remaining digits, then, looks much like what we already have:

⟨*integer producer, given first digit*⟩≡
```
(lambda (ts1)
  (sum-case ((more-digits) ts1)
    ((ds ts2) (⟨integer producer, given all digits⟩ ts2))
    ((msg ts2) (inr msg ts2))))
```

Finally, we have to return the answer. For this, we need an integer producer that represents a constant value (modulo free variables), an especially simple sort of computation:

[2] It may seem unnatural (no pun intended) to define `natural` as a nullary procedure instead of a value, but it will later take additional arguments and possibly become a macro.

⟨*integer producer, given all digits*⟩≡
```
(lambda (ts)
  (inl (string->number
         (list->string (cons d ds)))
       ts))
```

Naturally, the token stream is guaranteed to be unchanged in a simple computation.

Having completed the definition that handles the first production in the grammar, we move on to defining a procedure that handles the ⟨more digits⟩ non-terminal. More specifically, we define `more-digits` to be a nullary procedure—like `natural`—that gives us a producer. Whereas `natural` makes an integer-producing computation, `more-digits` instantiates a computation that produces a list of characters.

The grammar for ⟨more digits⟩ specifies two alternative productions: one like ⟨natural⟩ and one empty. Assuming that we want to absorb as many contiguous digits as possible into the number, we begin by trying the first alternative. If it fails, we accept the empty production (with the original token stream). Thus, `more-digits` begins this way:

⟨*long version of* `more-digits`⟩≡
```
(define more-digits
  (lambda ()
    (lambda (ts1)
      (sum-case (⟨list producer for more-digits⟩ ts1)
        ((ds ts2) (inl ds ts2))
        ((msg ts2) (⟨empty-list producer⟩ ts1))))))
```

Let's write the producer for the empty production first. It represents a constant-valued computation, similar to the one that returns the number in `natural`:

⟨*empty-list producer*⟩≡
```
(lambda (ts)
  (inl '() ts))
```

Most of the remaining code is identical to the body of `natural`, as it should be, considering that the grammar production is identical. The difference is in the return type:

⟨*list producer for* `more-digits`⟩≡
```
(lambda (ts1)
  (sum-case ((digit) ts1)
    ((d ts2)
     ((lambda (ts1)
        (sum-case ((more-digits) ts1)
          ((ds ts2)
           (⟨list producer, given all digits⟩ ts2))
          ((msg ts2) (inr msg ts2))))
      ts2))
    ((msg ts2) (inr msg ts2))))
```

Of course, one would usually β-reduce the inner `lambda` application, but we leave it in for consistency.

The code that returns the final value is like the corresponding code in `natural`, except that it does not convert the list of characters into a number:

⟨*list producer, given all digits*⟩≡
```
(lambda (ts)
  (inl (cons d ds) ts))
```

This completes the code for parsing natural numbers, as written by following the types rather blindly.

2.2 Becoming More Abstract

There were two distinct patterns in the code for `natural` and `more-digits`. One represents simple computations, like returning the empty list, the list of digits, or the integer value of such a list. In each case, the code looked like this:

⟨*producer pattern for returning an answer*⟩≡

```
(lambda (ts)
  (inl ⟨answer⟩ ts))
```

The other pattern was more complicated. It consisted of

1. invoking another producer,
2. receiving its return values (the result or error message and the new token stream),
3. checking for failure, and
4. either
   (a) sending the new token stream to a second producer, or
   (b) propagating the failure, skipping the second producer.

Abstracting over such code in the preceding section, the pattern looks like this:

⟨*producer pattern for sequencing two producers*⟩≡

```
(lambda (ts1)
  (sum-case (⟨producer #1⟩ ts1)
    ((⟨var⟩ ts2) (⟨producer #2⟩ ts2))
    ((msg ts2) (inr msg ts2))))
```

These two patterns correspond to the two operations used in monadic programming: `return` (also called *unit*) and `bind` (also called *monadic let*). As promised, we make coding patterns concrete by defining them as macros. Procedural definitions would be more conventional, but these macro definitions change in Section 4 to perform code rewrites that could not be accomplished with procedural abstractions.

Now `return` implements the simple answer-returning pattern:

⟨*implementation of the* `return` *pattern*⟩≡

```
(define-syntax return
  (syntax-rules ()
    ((return ?answer)
     (lambda (ts)
       (inl ?answer ts)))))
```

and `bind` implements the producer-sequencing pattern:

⟨*implementation of the* `bind` *pattern*⟩≡

```
(define-syntax bind
  (syntax-rules ()
    ((bind (?var ?producer1)
       ?producer2)
     (lambda (ts1)
       (sum-case (?producer1 ts1)
         ((?var ts2) (?producer2 ts2))
         ((msg ts2) (inr msg ts2)))))))
```

The type constructor $Producer$, together with `return` and `bind`, form a *Kleisli triple* [11]. (Actually, the third element of the Kleisli triple is not `bind`; it is `extend`, defined in Section 3.1. We find `extend` to be more convenient for mathematical manipulation and `bind` to be more convenient for monadic programming.) A Kleisli triple is equivalent to a monad; in fact, many authors drop the distinction altogether. Also, not all definitions for $Producer$, `return`, and `bind` form a Kleisli triple. The necessary properties are spelled out in detail in Section 3.

Using the monad operations, we can rewrite `natural` to be *much* more concise and readable:

⟨*definition of* `natural`⟩≡

```
(define natural
  (lambda ()
    (bind (d (digit))
      (bind (ds (more-digits))
        (return (string->number
                  (list->string
                    (cons d ds))))))))
```

The syntactic abstraction technique we just used appears repeatedly in the following sections: find a syntactic pattern, abstract it with a macro definition, and rewrite the original code more concisely using the macro definition.

One way to think about programming with `return` and `bind` is that the $Producer$ types form a family of abstract data types, and `return` and `bind` are the public operations that construct and combine producers. When we have a simple (non-producer) value and we want to instantiate a representation of a computation that produces that value, we use `return`. When we have representations for two computations and we want to sequence them, we use `bind` to construct a representation for the computation that feeds the result of the first into the second.

2.3 Monadic Combinators

We can write `more-digits` in a monadic style, but the patterns abstracted by `return` and `bind` do not completely absorb the code in `more-digits`. The part that checks to see if the first alternative failed, and if so proceeds to the second, does not fit either pattern.

⟨*unsatisfactory definition of* `more-digits`⟩≡

```
(define more-digits
  (lambda ()
    (lambda (ts1)
      (sum-case ((bind (d (digit))
                   (bind (ds (more-digits))
                     (return (cons d ds))))
                 ts1)
        ((ds ts2) (inl ds ts2))
        ((msg ts2) ((return '()) ts1))))))
```

While the code that implements alternate productions in a grammar does not fit the pattern of one of the core monad operations, it is clearly a pattern that will appear any time we need to check for the failure of one computation and perform another instead. Abstracting over the pattern gives us `orelse`, a *monadic combinator*:

⟨*unsatisfactory definition of* `orelse`⟩≡

```
(define-syntax orelse
  (syntax-rules ()
    ((orelse ?producer1 ?producer2)
     (lambda (ts1)
       (sum-case (?producer1 ts1)
         ((ds ts2) (inl ds ts2))
         ((msg ts2) (?producer2 ts1)))))))
```

If we rewrite `more-digits` one more time, using `orelse`, we get:

⟨*definition of* `more-digits`⟩≡

```
(define more-digits
  (lambda ()
    (orelse (bind (d (digit))
              (bind (ds (more-digits))
                (return (cons d ds))))
            (return '()))))
```

The definitions of both `natural` and `more-digits` now correspond very directly to the grammar for natural numbers. Furthermore, neither procedure deals explicitly with producer types except through `return` and `bind`.

We have, until now, simply assumed the existence of `digit`. Let's write it now. A call to `digit` creates a character producer that examines the first character in the token stream. If that character is numeric, it returns the character, "removing" it from the token stream. Otherwise, the computation fails and leaves the token stream unchanged:

⟨*unsatisfactory definition of* `digit`⟩≡

```
(define digit
  (lambda ()
    (lambda (ts)
      (if (or (null? ts)
              (not (char-numeric? (car ts))))
          (inr "not a digit" ts)
          (inl (car ts) (cdr ts))))))
```

(We represent our token streams in this article as lists of characters for simplicity.) Again, neither `return` nor `bind` helps simplify or clarify this code, because `digit` must access the token stream, which is not visible in procedures like `natural` that are written only in terms of the monadic operations.

3. Monads as Abstract Data Types

When we first introduced the $Producer$ type constructor, we presented it as an abstract means of representing computations by values. When we defined the `return` and `bind` operations, we provided a uniform interface to the abstraction. Ideally, all the other definitions would inhabit a space outside this abstraction boundary, even combinators like `orelse`. In the preceding section, though, we broke the $Producer$ abstraction in two ways.

First, in `orelse`, we took the results of producer expressions (constructed with `return` and `bind`, presumably) and applied them to token streams. This violation of the abstraction boundary is similar to taking a stack (a classic ADT) and performing a vector reference on it, just because we happen to know that the stack is represented as a vector. While our current representations for computations are, in fact, procedures that expect token streams, it is wrong for arbitrary code to assume such a representation. Instead, programmers need some explicit means of reifying computations as values of $Producer$ types in order to pass their own token streams (or whatever is appropriate to the specified representation types) to them and examine the results.

Second, in both `orelse` and `digit` we cobbled together arbitrary code—which happened to be of the proper type to generate $Producer$ values—and we expected to be allowed to treat those values as valid representations of computations. This violation of the abstraction boundary is similar to constructing our own vector to represent a stack and passing it to a procedure that expects a stack. This, too, is wrong. We did it because we needed to have access to the current token stream in the computation, but instead we need some explicit means of constructing a representation of a computation and reflecting it into the system so that it is accepted as something that has access to the threaded values.

The usual way to avoid violating the monad abstraction boundary is to move the offending operations—like `orelse` and `digit`—inside the boundary and treat them as fundamental monadic operators, having nearly the same status as `return` and `bind`. The weakness of such a solution is that it is often necessary to create operators like `digit` while writing a parser, not while creating a parser monad. A better solution is to create a small abstract data type for the monad and its most basic operators and to provide an interface for users of the monad to access the underlying representation of the monad (or at least a constructed view of it) in a limited way.

Monadic reflection, as introduced by Moggi [14] (though he does not use the phrase "monadic reflection") and amplified by Filinski [5], provides a means of crossing the monadic abstraction boundary with mathematically founded operators. Neither of these authors actually extends the idea of monadic reflection into the space of exposing and hiding representations in the sense of "reflective interpreters" and the like. Such an extension is new in this work, but related to the discussions by Chen and Hudak of monadic abstract data types [2].

3.1 Foundations

A monad consists of four things [12]:

1. a type constructor, T, for lifting a type α to a type that represents computations that produce values of type α,
2. a higher-order, polymorphic function (the *mapping function* of the monad) for lifting functions so that they take and return T types,

$$(\alpha \rightarrow \beta) \xrightarrow{map} (T(\alpha) \rightarrow T(\beta))$$

3. a polymorphic function (called the *unit* of the monad) for lifting a value of type α to the corresponding value of type $T(\alpha)$,

$$\alpha \xrightarrow{unit_\alpha} T(\alpha)$$

and

4. a polymorphic function (called the *multiplication* of the monad) for "un-lifting" a doubly-lifted value of type $T(T(\alpha))$ to the corresponding value of type $T(\alpha)$.

$$T(T(\alpha)) \xrightarrow{mult_\alpha} T(\alpha)$$

(In category theory, the first two elements of the monad are combined into a functor.) The possibility of iterating the T type constructor creates a sequence of "levels." The unit of the monad shifts up a level (more nesting or wrapping), and the multiplication shifts down (less nesting or wrapping). To guarantee that all the level shifting is coherent, the mapping function, unit, and multiplication must obey three equations:

$$\begin{aligned} mult_\alpha \circ map(unit_\alpha) &= id_{T(\alpha)} \\ mult_\alpha \circ unit_{T(\alpha)} &= id_{T(\alpha)} \\ mult_\alpha \circ map(mult_\alpha) &= mult_\alpha \circ mult_{T(\alpha)} \end{aligned}$$

A Kleisli triple for the monad consists of the type constructor, the unit (that is, `return`), and an *extension* operation:

$$(\alpha \rightarrow T(\beta)) \xrightarrow{extend_{\alpha,\beta}} (T(\alpha) \rightarrow T(\beta))$$

The `bind` form is simply a convenient notation for the common usage pattern of $extend$:

`((extend (lambda (v) N)) M)` = `(bind (v M) N)`

While it is possible to define the mapping function and multiplication of each monad directly, it is also possible to define both in terms of the `return` and `bind`. Only the indirect forms of the definitions follow.

For the $Producer$ type constructor we are using in our parsing examples, the mapping function—when applied to some procedure `f`—returns a procedure that takes a producer for one type and returns a producer for another. It uses `f` to get a value of the second type.

⟨*indirect definition of* `producer-map`⟩≡

```
(define producer-map
  (lambda (f)
    (lambda (producer)
      (bind (a producer)
        (return (f a))))))
```

The multiplication of the monad takes a value that represents a producer-producing computation. In other words, when it is applied to a token stream, it either fails or returns a producer and a new token stream. We can use `bind` for a very concise definition, and write `mult` this way:

⟨*indirect definition of* `mult`⟩≡

```
(define mult
  (lambda (producer-producer)
    (bind (producer producer-producer)
      producer)))
```

The unit of the monad is actually the same thing as `return`:

⟨*indirect definition of* `unit`⟩≡

```
(define unit
  (lambda (a)
    (return a)))
```

We see, then, that a monad can be defined completely in terms of a Kleisli triple. The equivalence is bidirectional; we shall not demonstrate it here, but the Kleisli triple can be defined in terms of the monad, too.

3.2 Monadic Reflection

If Kleisli triples and monads are equivalent, why would we choose one over the other? As was evident in Section 2.2, Kleisli triples are excellent tools for monadic-style programming. That is to say, they provide an appropriate means of abstractly manipulating the values that we use to represent computations.

The unit and multiplication of a monad, on the other hand, succeed in just the place where Kleisli triples failed. They provide the appropriate means for crossing the monadic abstraction boundary via level-shifting. In other words, the `unit` and `mult` are excellent tools for *monadic reflection*.

In order to talk about "clean" reflective level crossings, it is necessary to have some notion of *opaque* and *transparent* types. A simple mathematical understanding of the definition of $Producer$

$$Producer(\alpha) = Tokens \rightarrow (\alpha + ErrMsg) \times Tokens$$

treats the two sides of the equation as synonyms. From a software engineering perspective, however, there is a significant difference between the type constructor being defined and the body of its definition. To exploit this difference, let us rewrite the types of `unit` and `mult`, treating the outermost level as opaque and the inner levels as transparent whenever there are nested applications of the type constructor. They become

$$P(\alpha) \xrightarrow{unit_{P(\alpha)}} P(T(\alpha))$$

and

$$P(T(\alpha)) \xrightarrow{mult_\alpha} P(\alpha)$$

where P represents an opaque version of T. Using these types, the outer "interface" of the type always remains opaque. The types for `return` and `extend` (and thus `bind`) refer only to the opaque version of the type constructor:

$$\alpha \xrightarrow{return_\alpha} P(\alpha)$$

and

$$(\alpha \rightarrow P(\beta)) \xrightarrow{extend_{\alpha,\beta}} (P(\alpha) \rightarrow P(\beta))$$

It might seem that these operations allow no means of "reaching through" the opaque type to do anything interesting with the transparent version, but in fact, they provide plenty of power when the operations are used in conjunction with each other.

Let us return to our unsatisfactory definitions of `digit` and `orelse` to see how judicious use of `unit` and `mult` create clean and explicit abstraction-boundary crossings. We begin with `digit`, where we want to construct a representation for a non-standard computation (i.e., one that cannot be constructed by `return` or `bind`). Furthermore, we want our hand-constructed procedure to be accepted as a valid digit (numeric character) producer. Here is the code that we want to act as a digit producer; it is taken straight from the old definition of `digit`:

⟨*custom digit producer*⟩≡
```
(lambda (ts)
  (if (or (null? ts)
          (not (char-numeric? (car ts))))
      (inr "not a digit" ts)
      (inl (car ts) (cdr ts))))
```

Just as we do for `42` or `(car '(1 2 3))`, we use `return` to construct a computation that produces this value:

⟨*digit-producer producer*⟩≡
```
(return ⟨custom digit producer⟩)
```

Finally, we use `mult` to "shift down a level." That is, `mult` will turn the digit-producer producer into a plain digit producer, explicitly coercing our hand-constructed value into a valid instance of the abstract data type.

⟨*definition of* `digit`*, using* `mult`⟩≡
```
(define digit
  (lambda ()
    (mult ⟨digit-producer producer⟩)))
```

Although `orelse` is longer and more complicated, the same kind of techniques work for rewriting it in a more satisfactory style. This time, we use both `unit` and `mult`, because `orelse` needs to shift up (lift the representation of the underlying computation into a value the user can manipulate) as well as down. We begin by lifting both of the incoming producers:

⟨*definition of* `orelse`*, using* `unit` *and* `mult`⟩≡
```
(define-syntax orelse
  (syntax-rules ()
    ((orelse ?producer1 ?producer2)
     (bind (p1 (unit ?producer1))
       (bind (p2 (unit ?producer2))
         ⟨producer that performs alternation⟩)))))
```

As in `digit`, we need a producer that cannot be written using `return` and `bind`, so we construct one by hand and use `mult` to reflect it into the system:

⟨*producer that performs alternation*⟩≡
```
(mult (return (lambda (ts1)
                (sum-case (p1 ts1)
                  ((ds ts2) (inl ds ts2))
                  ((msg ts2) (p2 ts1))))))
```

The difference between this code and what appeared in the body of the original version of `orelse` is that we have used `p1` and `p2` in place of the producers to which `orelse` was applied. Explicitly applying `p1` and `p2` to token streams is a valid thing to do, because `unit` yields transparent values wrapped in an opaque coating, and `bind` strips away the coating.

3.3 Abstracter and Abstracter

Just as `return` and `bind` are syntactic abstractions of the patterns for simple construction and sequencing of producer values, we can formulate patterns that abstract the common usage of `unit` and `mult`. We assert that, if we were to go out and write hundreds of procedures using `unit` and `mult`, we would see the same patterns over and over: the ones used in `digit` and `orelse`. The pattern for using `unit` looks like this:

⟨*producer pattern for reifying a producer*⟩≡
```
(bind (⟨var⟩ (unit ⟨producer #1⟩))
  ⟨producer #2⟩)
```

And whenever we use `mult`, we apply `return` to a `lambda` expression:

⟨*producer pattern for reflecting a constructed producer*⟩≡
```
(mult (return (lambda (⟨var⟩)
                ⟨expression⟩)))
```

The effect of these compositions is even more evident when the constituent operations are written as arrows. Assume that ⟨*producer #1*⟩ has opaque type $P(\alpha)$ but ⟨*producer #2*⟩ treats ⟨*var*⟩ as the transparent $T(\alpha)$, returning a value of opaque type $P(\beta)$. In terms of `extend`, this means that the body is like a function

$$T(\alpha) \xrightarrow{g} P(\beta)$$

and the whole reification composition is:

$$P(\alpha) \xrightarrow{unit_{P(\alpha)}} P(T(\alpha)) \xrightarrow{extend_{T(\alpha),\beta}(g)} P(\beta)$$

The reflection composition yields a simple conversion from transparent to opaque types:

$$T(\alpha) \xrightarrow{return_{T(\alpha)}} P(T(\alpha)) \xrightarrow{mult_{\alpha}} P(\alpha)$$

As is our wont, we turn these patterns into macros. The first we call `reify`:

⟨*definition of* `reify`⟩≡
```
(define-syntax reify
  (syntax-rules ()
    ((reify (?var ?producer1)
       ?producer2)
     (bind (?var (unit ?producer1))
       ?producer2))))
```

The second we call `reflect`:

⟨*definition of* `reflect`⟩≡
```
(define-syntax reflect
  (syntax-rules ()
    ((reflect (?var) ?expression)
     (mult
       (return (lambda (?var) ?expression))))))
```

Effectively, `reflect` exposes the threaded token stream to the expression in its body.

We can now use `reflect` to simplify `digit` one more time:

⟨*definition of* `digit`⟩≡
```
(define digit
  (lambda ()
    (reflect (ts)
      (if (or (null? ts)
              (not (char-numeric? (car ts))))
          (inr "not a digit" ts)
          (inl (car ts) (cdr ts))))))
```

Using `reflect` and `reify` together, we get a new definition of `orelse`:

⟨*definition of* `orelse`⟩≡
```
(define-syntax orelse
  (syntax-rules ()
    ((orelse ?producer1 ?producer2)
     (reify (p1 ?producer1)
       (reify (p2 ?producer2)
         (reflect (ts1)
           (sum-case (p1 ts1)
             ((ds ts2) (inl ds ts2))
             ((msg ts2) (p2 ts1)))))))))
```

These are our final definitions of `digit` and `orelse`. They are now completely explicit in their crossings of abstraction boundaries. Also, the representation of computations is remarkably abstract. We need know only that producers can be applied to token streams and that they return a sum value and a new token stream. We never use `lambda` to construct producers directly.

3.4 A Grammar for Monadic Programming

When we decried the original code for `digit` and `orelse`, we were appealing to what we hoped was a shared implicit intuition, which we now make explicit. What is it that makes us uncomfortable with the following code?

⟨*bad code*⟩≡
```
(bind (x (natural))
  (lambda (ts)
    (inl (+ x 2) (cdr ts))))
```

What bothers us is that we expect the body of the `bind` expression to be another `bind` or a `return`, or maybe a `reify` or a `reflect`, but certainly not a `lambda`. In other words, programs written in a "monadic style" are really written in a particular sublanguage in which only certain forms are allowable.

We make the language of monadic programming explicit by presenting a grammar for it. This grammar requires both the right-hand side and the body of `bind` expressions to be other monadic expressions, and so on.

⟨program⟩ $\rightarrow D \ldots$ (`run` M E)
$D \rightarrow$ (`define` V_M R)
$R \rightarrow$ (`lambda+` ($V \ldots$) M)
$M \rightarrow$ (`return` E)
  | (`bind` (V M) M)
  | (`reflect` (V) E)
  | (`reify` (V M) M)
  | (V_M $E \ldots$)
  | derived monadic expression
$E \rightarrow$ arbitrary Scheme expression

By "derived monadic expression," we mean user-defined syntactic forms—like `orelse`—that expand into monadic expressions. By "arbitrary Scheme expression," we mean code that does *not* contain monadic subexpressions.

The relationships among `return`, `bind`, `reflect`, and `reify` might be better understood by examining typing rules for them. The rules in Figure 1, for the sake of brevity, abbreviate $Producer$ as P. No rules are given for arbitrary expressions E. Instead, these four rules are meant to augment the typing rules for standard expressions.

There are two additional forms introduced in this grammar: `run` and `lambda+`. Without `lambda+`, there would be no "roots" for the portion of the grammar that deals with monadic expressions, nowhere to get started with monadic programming. For now, we let `lambda+` be synonymous with `lambda`. To conform to this grammar, `digit`, `natural`, and `more-digits` should be modified to use `lambda+`.

The `run` form simply starts a computation by passing the initial token stream (or other store-like value) to a producer:

⟨*definition of* `run`⟩≡
```
(define-syntax run
  (syntax-rules ()
    ((run ?producer ?exp)
     (?producer ?exp))))
```

For example, this use of `run`:

```
(run (natural) (string->list "123abc"))
```

would run our natural-number parsing program and return `123` (left-injected) and the remaining characters `(#\a #\b #\c)`.

4. Optimizing Monadic Programs

With both the parsing operators like `digit` and the simple client code like `natural` written in terms of `return`, `bind`, `reflect`, and `reify`, the inner abstraction boundary around the monad is satisfyingly small. The performance, though, is inadequate for use in a real compiler or interpreter. The largest source of overhead expense is all the closure creation, which a compiler may or may not eliminate. To provide a stronger guarantee than "we hope the compiler cleans this up for us," it is possible to create new closure-free versions of the macros for the core operators.

$$(\mathit{return})\quad \frac{\Gamma \vdash E : \tau}{\Gamma \vdash (\mathtt{return}\, E) : P(\tau)}$$

$$(\mathit{bind})\quad \frac{\Gamma \vdash M_1 : P(\tau_1) \qquad \Gamma, v\!:\!\tau_1 \vdash M_2 : P(\tau_2)}{\Gamma \vdash (\mathtt{bind}\,(v\, M_1)\, M_2) : P(\tau_2)}$$

$$(\mathit{reify})\quad \frac{\Gamma \vdash M_1 : P(\tau_1) \qquad \Gamma, v\!:\!(S \to (\tau_1 + \mathit{ErrMsg}) \times S) \vdash M_2 : P(\tau_2)}{\Gamma \vdash (\mathtt{reify}\,(v\, M_1)\, M_2) : P(\tau_2)}$$

$$(\mathit{reflect})\quad \frac{\Gamma, v\!:\!S \vdash E : (\tau + \mathit{ErrMsg}) \times S}{\Gamma \vdash (\mathtt{reflect}\,(v)\, E) : P(\tau)}$$

Figure 1. Typing Rules

Let's look at the expansion of a small part of our natural number parser, the first of the alternatives in `more-digits`:

⟨`more-digits` *fragment*⟩≡

```
(bind (ds (more-digits))
  (return (cons d ds)))
```

Using the most recent versions of `bind` and `return`, this code expands into:

⟨`more-digits`-*fragment expansion*⟩≡

```
(lambda (ts1)
  (sum-case ((more-digits) ts1)
    ((ds ts2) ((lambda (ts)
                 (inl (cons d ds) ts))
               ts2))
    ((msg ts2) (inr msg ts2))))
```

In the expansion, every subexpression that denotes a producer value, be it a call like `(more-digits)` or a `lambda` expression, is applied to a token stream. This property will hold in all such programs, as it is guaranteed by our grammar.

4.1 Eliminating the Closures

According to the implementation from the preceding sections, every producer expression will construct a closure, either directly (by expanding into a `lambda` expression) or indirectly (by invoking a procedure that returns a closure). These closures are then immediately applied to token streams. Of course, the direct expansion into `lambda` and immediate application (as in the preceding example) becomes `let` in nearly every Scheme implementation, but the sites where closures are returned by procedure calls are much harder for a compiler to optimize. One way to improve both the memory and space use of the code is to remove the need for the two-stage application. Since, in the expansion, the token stream is always available to finish off the application, we never need to partially apply procedures like `digit`. Instead, we can modify the definitions of our monadic-programming macros so the token stream is passed as an extra argument to the existing procedures.

The `lambda+` form, which we introduced in the preceding section, is the starting point for the extra arguments:

⟨*improved definition of* `lambda+`⟩≡

```
(define-syntax lambda+
  (syntax-rules ()
    ((lambda+ (?formal ...) ?body)
     (lambda (?formal ... ts)
       ⟨body of token-accepting function⟩))))
```

We now need to thread the token-stream argument appropriately into the body. Since we know that this body must be a monadic expression, we need only change the implementation of those forms consistently with the new "un-curried" `lambda+` form.

The simplest case is if the body is an application of a user-defined procedure, such as a call to `digit`. In this case, we need to make sure to thread our store through as the last argument to the call. We accomplish this with the helper form `with-args`:

⟨*definition of* `with-args`⟩≡

```
(define-syntax with-args
  (syntax-rules ()
    ((with-args (?extra-arg ...)
                (?operator ?arg ...))
     (?operator ?arg ... ?extra-arg ...))))
```

It may seem that `with-args` is more general than necessary, since it can handle multiple extra arguments, but this generality offers us a great deal of leverage, as we shall see later. Using `with-args`, we can finish the definition of `lambda+` like this:

⟨*body of token-accepting function*⟩≡

```
(with-args (ts) ?body)
```

This code is well-formed only if the body is in the form of an operator and some arguments. If we look back at the grammar, we see that this is indeed the case.

The definitions of `bind` and `return` must now handle extra input in their patterns. In `bind`, these extra arguments must be threaded into the subforms:

⟨*improved definition of* `bind`⟩≡

```
(define-syntax bind
  (syntax-rules ()
    ((bind (?var ?rhs) ?body ?ts ...)
     (sum-case (with-args (?ts ...) ?rhs)
       ((?var ?ts ...)
        (with-args (?ts ...) ?body))
       ((msg ?ts ...) (inr msg ?ts ...))))))
```

The token-stream parameter(s) used in the right-hand side are the same ones (i.e., the same names as those) bound by `let-values` in the body. We need not worry about shadowing, though, since the token stream is necessarily threaded, and there can be no free references to it in the body.

In `return`, the extra arguments need to be threaded back out, along with the desired return value.

⟨*improved definition of* `return`⟩≡

```
(define-syntax return
  (syntax-rules ()
    ((return ?answer ?ts ...)
     (inl ?answer ?ts ...))))
```

Thus, `return` becomes an alias for `inl`, as it should be.

Since we no longer run a computation by first evaluating it and then passing the result a token stream, we must modify `run` to follow the new protocol:

⟨*improved definition of* `run`⟩≡

```
(define-syntax run
  (syntax-rules ()
    ((run ?producer ?exp ...)
     (with-args (?exp ...) ?producer))))
```

The new version converts the initial stream(s) into argument(s) to the producer. The grammar in the preceding section supported only a single "hidden" argument. In order for it to support the generality that is included in the new versions of these operators, it should be modified to allow additional arguments to `run`. The same sort of modification is necessary in the grammar rule for `reflect`. It should allow additional variables to be bound to the current values of the additional store-like parameters.

The `reflect` and `reify` forms require a bit more analysis before they can be optimized. We begin with `reflect`. There are two ways to proceed here. One is to recognize that while the added syntax we have imposed with `reflect` is good for software engineering, the `reflect` form is still mathematically equivalent to what we started with: a directly constructed `lambda` expression for a producer. (This mathematical equivalence, which comes from the monad equations, is a good thing. It validates our sequence of abstractions and transformations.) The other approach is simply to begin with the macro definition for `reflect` and follow all the definitions and β-reductions, eventually concluding that `reflect` is merely an alias for `lambda`. Either way, the result is the same. Applying a `reflect` form to a token stream is the same as applying the corresponding `lambda` expression. In other words, under our new protocol, `reflect` expands into a `let`.

⟨*improved definition of* `reflect`⟩≡

```
(define-syntax reflect
  (syntax-rules ()
    ((reflect (?var ...) ?expression ?ts ...)
     (let ((?var ?ts) ...)
       ?expression))))
```

We have carried the potential for threading multiple values through `reflect`, just as we did for `with-args`. This generalizes the version of `reflect` in the preceding sections. Of course, the `let` we just introduced merely renames the token-stream parameter(s).

More mechanism is required to implement `reify` well. If we continue to reify computations as values, using the threaded functional representations, we must pay for first-class procedures:

⟨*improved definition of* `reify`*, first try*⟩≡

```
(define-syntax reify
  (syntax-rules ()
    ((reify (?var ?rhs) ?body ?ts ...)
     (let ((?var (lambda (?ts ...)
                   (with-args (?ts ...) ?rhs))))
       (with-args (?ts ...) ?body)))))
```

While this works, it creates the first-class procedures we were trying to avoid. The point of `reify` is to allow the code in the body to poke at the reified producer by passing it token streams and examining the results explicitly. We can support this functionality without forming a closure by constructing the expansion-time equivalent of a locally-applicable closure: a local macro. We bind (at compile time) the variable to a syntax transformer that generates the right code:

⟨*improved definition of* `reify`⟩≡

```
(define-syntax reify
  (syntax-rules ()
    ((reify (?var ?rhs) ?body ?ts ...)
     (let-syntax
         ((?var (syntax-rules ()
                  ((?var ?ts ...)
                   (with-args (?ts ...) ?rhs)))))
       (with-args (?ts ...) ?body)))))
```

This new definition has a certain constraint that was not present in the procedural version: the bound variable must appear in the `?body` only in operator position. This is due, in part, to the inability to do macro-like replacement of plain identifiers in Scheme's standardized syntactic extension mechanisms,[3] but the restriction boosts efficiency anyway. It prevents us from leaking unwanted computational effort into the runtime.

The new definition of `reify` is backed by a mathematical equivalence, too. The original definition of `reify` was mathematically equivalent (again by the monad equations) to

[3] Some implementations, such as Chez Scheme [3], do support substitution for all identifiers in the scope of the macro binding.

substituting the right-hand side for the variable in the body. Our new definition does just this.

4.2 The Closure-Free Expansion

Using the new definitions for `return`, `bind`, etc., we get wonderfully improved expansions for monadic programs. For instance, the fragment of code at the beginning of this section, which used to contain five different closure-creation sites, now expands into the following:

⟨`more-digits`-*fragment expansion, improved*⟩≡

```
(sum-case (more-digits ts)
  ((ds ts) (inl (cons d ds) ts))
  ((msg ts) (inr msg ts)))
```

The new code creates no closures at all. The lack of rampant anonymous procedures also makes the new code much more amenable to compiler optimizations. For example, if all the code for parsing is put in a single mutually recursive block (i.e., a single `letrec`), we would expect a good compiler to turn all the calls into direct calls to known code addresses.

4.3 Alternative Sum-Type Representations

The representation we have used for sum-type values requires a dispatch at every return site (see the appendix). There are two useful alternatives to this approach.

One alternative is simply to return no value for failure, and one value for success. This is no faster in the abstract than returning a boolean value, since there remains a dispatch at every return site, but some implementations of Scheme provide especially fast ways to dispatch on argument count [4]. Thus, while this technique does not decrease the number of steps, it may decrease the absolute running time of the program.

The second alternative is the only one that really eliminates the return-site dispatch. One provable property of our monad definition is that, in the absence of reification, failures are propagated up through the entire extent of the computation. In other words, it is only in operators like `orelse` that failures may be caught and acted upon. We could capture a continuation at each such dispatch point and pass it down into the subcomputations. When we want to signal a failure (as in `digit`), we invoke the most recently captured continuation. This is close in both spirit and theory to the direct-style monadic programming of Filinski [5]. In this implementation, no checks have to be made at each normal return point, but the overhead for continuation creation may outweigh this savings. (Actually, this technique does not require full continuations; it needs only escapes, which may be implemented more cheaply than full first-class continuations.)

Naïvely implemented parsing routines, like the one we wrote for natural numbers, will make heavy use of `orelse`. Thus, depending on the expense of the second alternative, it may not be worthwhile. On the other hand, if a grammar is made very deterministic through the use of pre-calculation (of "first" and "follow" sets, for example), then failures may be truly exceptional, and the continuation-based alternative could eliminate a significant amount of overhead.

5. Conclusions

The example in this paper has been exclusively about parsing, but the results extend across a much broader scope: any composition of store-like monads, possibly composed with an error or lifting monad. The macros in the preceding section are defined in such a way that it is easy to support the threading of multiple store-like parameters through computations. In fact, the only form that must be changed to add a parameter is `lambda+`. For example, if we want to thread three stores through the computation, we rewrite `lambda+` this way:

⟨*definition of* `lambda+` *with 3 stores*⟩≡

```
(define-syntax lambda+
  (syntax-rules ()
    ((lambda+ (?formal ...) ?body)
     (lambda (?formal ... s1 s2 s3)
       (with-args (s1 s2 s3) ?body)))))
```

The use of `with-args` in all the other forms will drive them to expand in ways that propagate the store parameters correctly. With our current definitions, any user-level code that uses `reflect` must be rewritten to accept the extra store parameters, and any code that uses `reify` must apply the reified values to additional arguments. One way that this work could be extended is to implement a mechanism by which user-level code would be able to refer to only those "hidden" parameters that they need to see at any point. This is possible with more sophisticated macros.

At the end of Section 4.3 we alluded to the possibility of preprocessing the grammar and/or parser to boost its performance. Another possible direction we see for research in this area is to combine the "fast LR parsing via partial evaluation" techniques of Sperber and Thiemann [17] with our expansion-time optimizations. The primary goal of most functional parsing research is to make parsers easier for *people* to write, but the same results should simplify the work of parser generators.

Even if our goal had been to compile monadic programs directly into a lower-level language, the more rigorous style afforded by explicit monadic reflection would make the compilation process more tractable. For example, a typical parser written in Haskell or Scheme will be much easier to convert to C without arbitrary anonymous functions in the user code, which the user expects to be treated as representations of computations.

The measurable performance benefit from the optimized (store-threaded) macros varies depending on the Scheme implementation. One production-grade parser that uses the macros from this article is used to parse a kind of annotated table-definition language for databases. The parser is split into modules that do lexical analysis and phrasal analysis, with the output of the first serving as the token stream for the second. One of the regular inputs to this parser contains about 150 tables, at a total file length of about 3000 lines. Running on Chez Scheme [3], the total time to parse the input and construct the parse tree is less than 2 tenths of a second on typical personal computer hardware. There is no measurable difference between the different versions of the macros, implying that Chez Scheme is already eliminating all the overhead that might be introduced by closure creation, even across procedure calls. Running on DrScheme [15, 6], the total parse time on the same hardware is about 1.5 seconds. There is a 10% to 12% decrease in the parse time using the improved macros from Section 4.

Thus, the benefits of following a grammar for monadic programming—even for operators that depend somewhat on the monad's representation—are two-fold: First, the programs written in a stricter monadic style are more elegant, less *ad hoc*. While it is possible to write well-typed monadic programs without using explicit reflection operators, they violate abstractions in the same ways that ill-typed (but runnable) programs do in C when they cast a file pointer to be an integer and add 18 to it, just because some programmer happens to know that the result will be meaningful. Second, the rigor that makes programs *feel* better can also make them *run* better. While a sufficiently "smart" compiler or partial evaluator might eliminate the closure overhead just as well as our rewritten operators, there is an element of certainty that comes from shifting the work even earlier than compile time. By making sure that the optimization happens at expansion time, we depend less on the the analysis phase of a compiler and more on our own mathematics.

Acknowledgments

The authors appreciate the comments and recommendations provided by anonymous referees. Kevin Millikin was involved in many discussions as the paper was initially written. Michael Sperber offered several valuable suggestions, including the recommendation that this work be shared in the context of the Scheme Workshop. A special thanks goes to Mitch Wand for extensive comments, careful readings, and pointed demands for a better treatment of the types of the basic operators.

References

[1] Richard Bird. *Introduction to Functional Programming Using Haskell*. Prentice Hall Series in Computer Science. Prentice Hall Europe, second edition, 1998.

[2] Chih-Ping Chen and Paul Hudak. Rolling your own mutable ADT: A connection between linear types and monads. In *Conference Record of POPL '97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, Paris, France, January 1997. ACM Press.

[3] R. Kent Dybvig. *Chez Scheme Version 7 User's Guide*. Cadence Research Systems, 2005.

[4] R. Kent Dybvig and Robert Hieb. A new approach to procedures with variable arity. *Lisp and Symbolic Computation*, 3(3):229–244, 1990.

[5] Andrzej Filinski. Representing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, New York, January 1994. ACM Press.

[6] Matthew Flatt. PLT MzScheme: Language manual. `http://download.plt-scheme.org/doc/mzscheme/`, 2005.

[7] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[8] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[9] Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[10] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998.

[11] Heinrich Kleisli. Every standard construction is induced by a pair of adjoint functors. In *Proceedings of the American Mathematical Society*, volume 16, pages 544–546, 1965.

[12] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, second edition, 1998.

[13] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, April 1989.

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

[15] PLT. PLT DrScheme: Programming environment manual. `http://download.plt-scheme.org/doc/drscheme/`, 2005.

[16] David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[17] Michael Sperber and Peter Thiemann. The essence of LR parsing. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 146–155, La Jolla, 1995. ACM Press.

[18] Philip Wadler. How to replace failure by a list of successes. In *Second International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. Springer-Verlag.

[19] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, December 1992.

Appendix

As long as sum-type values never need to be stored in data structures (and they do not, in this article), they can be represented efficiently as “tagged” multiple values. The tag is simply `#t` for left-injected values:

⟨*tag-based* `inl`⟩≡

```
(define-syntax inl
  (syntax-rules ()
    ((inl ?arg ...)
     (values #t ?arg ...))))
```

and `#f` for right-injected values:

⟨*tag-based* `inr`⟩≡

```
(define-syntax inr
  (syntax-rules ()
    ((inr ?arg ...)
     (values #f ?arg ...))))
```

For “casing” sum-type values, we use a new syntactic form `sum-case`, as demonstrated in the following example:

⟨*sum type example*⟩≡

```
(define add1-or-zero
  (lambda (thunk)
    (sum-case (thunk)
      ((n) (+ n 1))
      ((z) 0))))

(list (add1-or-zero (lambda () (inl 42)))
      (add1-or-zero (lambda () (inr 0))))
```

The last expression evaluates to the list `(43 0)`.

Defining a macro for `sum-case` is relatively straightforward in a Scheme implementation that has a direct means of generating temporary variables in macros. The portable version of the macro is made much more complicated by the need to generate a list of temporaries:

⟨*portable tag-based* `sum-case`⟩≡

```
(define-syntax sum-case
  (syntax-rules ()
    ((sum-case ?exp
       ((?left-var ...) ?left-result)
       ((?right-var ...) ?right-result))
     (gen-var-list (?left-var ...)
       (sum-case-help () ?exp
         ((?left-var ...) ?left-result)
         ((?right-var ...) ?right-result))))))

(define-syntax sum-case-help
  (syntax-rules ()
    ((sum-case-help (?temp ...) ?exp
       ((?left-var ...) ?left-result)
       ((?right-var ...) ?right-result))
     (call-with-values (lambda () ?exp)
       (lambda (tag ?temp ...)
         (if tag
             (let ((?left-var ?temp) ...)
               ?left-result)
             (let ((?right-var ?temp) ...)
               ?right-result)))))))
```

```
(define-syntax gen-var-list
  (syntax-rules ()
    ((gen-var-list ()
       (?head (?y ...) ?tail ...))
     (?head (?y ...) ?tail ...))
    ((gen-var-list (?v0 ?v ...)
       (?head (?y ...) ?tail ...))
     (gen-var-list (?v ...)
       (?head (?y ... temp) ?tail ...)))))
```