Monoids model extensibility

or, Moxy: a language with extensibly extensible syntax

Michael Arntzenius
Carnegie Mellon University
daekharel@gmail.com

Abstract

Many languages, libraries, or systems advertise themselves as extensible, but these claims are usually informal. We observe that *monoids* formally characterize much of the essence of extensibility. We present extensibility monoids for several pre-existing systems, including grammars, macros, and monad transformers. To show the utility of this approach, we present Moxy, a syntactically extensible programming language. Moxy's implementation is surprisingly simple for the power it offers. We hope the use of monoids as an organizing principle of extensibility will offer similar utility in other domains.

Keywords Extensibility, extensible languages, macros, monoids, parser-combinators, parsing, syntax.

1. Introduction

The concept of extensibility recurs frequently in programming languages research, but lacks a concrete definition. The most one can say is "you know it when you see it". This paper observes that *monoids* capture some of the essential properties of extensibility. In Section 2, several common examples of extensibility are fruitfully analysed in terms of their monoids.

To show this approach's utility on a larger scale, in Section 3 we present Moxy, a language designed to be syntactically extensible — to permit the programmar to add syntax for new data types, operations, embedded DSLs, and so on — by using monoids to abstract extensibility. Moxy's design has several interesting qualities:

1. Moxy is not a Lisp or Scheme; extensibility is not achieved by the sacrifice of syntactic conveniences such as infix operators.[1]
2. Moxy is *self-extensible*: extensions are written in Moxy itself.
3. Moxy extensions are *modular*, that is to say, they are imported just like ordinary library definitions.
4. Moxy extensions are *scoped*; an extension can be imported within a whole file, within a module, or just within a single **let**-expression.
5. Moxy extensions are not limited to adding new forms of *expression*. They can also add new forms of *pattern* for use in pattern-matching, for example.
6. Moxy is *meta-extensible*: extensions can *themselves* be extended. For example, a Moxy programmer could create an embedded DSL as an extension library, and leave that DSL open to further extension by future programmers.
7. Moxy is *homogenously extensible*: All forms of extension — new expressions, new patterns, extensions to some library-defined extensible extension, and so forth — are accomplished by the same mechanism; no special privilege is given to built-in forms of extension, or to expressions over patterns.
8. Finally, Moxy itself is simple: about 2,000 lines of Racket code.

[1] Opinions on the desirability of this property may differ.

2. Monoids as a framework for extensibility

A monoid $\langle A, \cdot, \varepsilon \rangle$ consists of a set A, an operation $(\cdot) : A \times A \to A$, and an element $\varepsilon : A$ satisfying:

$$\begin{array}{rcll} a \cdot (b \cdot c) & = & (a \cdot b) \cdot c & \text{associativity} \\ a \cdot \varepsilon & = & a & \text{right-identity} \\ \varepsilon \cdot a & = & a & \text{left-identity} \end{array}$$

A notion of extensibility may be defined by giving a monoid where A is the set off all possible extensions, $(\cdot)$ is an operator that *combines* or *composes* two extensions, and ε is the "null extension", representing the lack of any additional behavior.

We show that monoids are a good model for extensibility by analysing five distinct forms of extensibility into their respective monoids: composition of context-free grammars, lisp-style macros, open recursion, middleware in a web framework, and monad transformers.

2.1 Composition of context-free grammars

Consider the following grammar for a λ-calculus:

$$\begin{array}{lrcl} \textbf{expressions} & e & ::= & x \mid \lambda x.\, e \mid e\, e \\ \textbf{variables} & x & ::= & \text{... (omitted) ...} \end{array}$$

It is easy enough to add infix arithmetic to this language:

$$\begin{array}{lrcl} \textbf{expressions} & e & ::= & x \mid \lambda x.\, e \mid e\, e \\ & & \mid & n \mid e \otimes e \\ \textbf{numerals} & n & ::= & d \mid dn \\ \textbf{digits} & d & ::= & 0 \mid 1 \mid \ldots \mid 9 \\ \textbf{operators} & \otimes & ::= & + \mid - \mid \times \mid / \end{array}$$

It is similarly easy to add lists:

$$\begin{array}{lrcl} \textbf{expressions} & e & ::= & x \mid \lambda x.\, e \mid e\, e \\ & & \mid & [\,] \mid [es] \\ \textbf{expression lists} & es & ::= & e \mid e, es \end{array}$$

Formally, each of these three grammars is a separate, complete object. But there is a clear sense in which the two latter languages are composed out of parts: "λ-calculus *plus* infix arithmetic", "λ-calculus *plus* lists". And, without even writing its grammar, it is obvious there exists a fourth language: "λ-calculus *plus* infix arithmetic *plus* lists". Can we formalize this colloquial "*plus*" operator? We can; and it is even a monoid!

Context-free grammars. A context-free grammar (CFG) is a tuple $\langle \Sigma, N, P, e\rangle$, where Σ is a finite set of terminal symbols, N is a set (disjoint from Σ) of nonterminal symbols, P is a finite set of *production rules*, and $e \in N$ is an initial non-terminal. Production rules $p \in P$ have the form $N \hookrightarrow (\Sigma \cup N)^*$; that is, $P \subseteq N \times (\Sigma \cup N)^*$.

Fix a terminal set Σ, a (possibly infinite) nonterminal set N, and an initial nonterminal e. That is, consider grammars over a known alphabet (e.g. Unicode characters). The non-terminal set is also fixed, but this is no great limitation, since it is permitted to be infinite; we will consider nonterminals to come from an inexhaustible set of abstract names. And we fix an initial nonterminal e, intended to represent well-formed expressions in some language.

Having fixed all this, a grammar is fully specified by a finite set of production rules P. Our original λ-calculus is represented by:

$$\begin{aligned} P_\lambda \quad = \quad \{ & e \hookrightarrow x, \\ & e \hookrightarrow \lambda x.e, \\ & e \hookrightarrow e\,e, \\ & \text{... (omitted rules for } x\text{) ...}\} \end{aligned}$$

Composing CFGs. To compose two grammars represented as production-rule-sets, simply take their union. For example, consider the sets of rules *added* when extending our language with infix arithmetic and lists respectively:

$$\begin{aligned} P_\otimes \quad &= \quad \{e \hookrightarrow n,\ e \hookrightarrow e \otimes e, \\ &\qquad n \hookrightarrow d,\ n \hookrightarrow dn, \\ &\qquad d \hookrightarrow 0,\ \ldots,\ d \hookrightarrow 9, \\ &\qquad \otimes \hookrightarrow +,\ \otimes \hookrightarrow -,\ \otimes \hookrightarrow \times,\ \otimes \hookrightarrow /\} \\ P_{[]} \quad &= \quad \{e \hookrightarrow [\,],\ e \hookrightarrow [es], \\ &\qquad es \hookrightarrow e,\ es \hookrightarrow e\mathbf{,}\, es\} \end{aligned}$$

If we take $P_\lambda \cup P_\otimes$, it gives us precisely the grammar of our second language, "λ-calculus *plus* infix arithmetic". $P_\lambda \cup P_{[]}$ is our third language, "λ-calculus *plus* lists". And $P_\lambda \cup P_\otimes \cup P_{[]}$ is the hypothesized fourth language, "λ-calculus *plus* infix arithmetic *plus* lists".

We have thus successfully separated our languages into parts — one which represents λ-calculus, one which represents infix arithmetic, one which represents lists — and found an operator that can combine them again. Finally, note that this operator (union of production-rule-sets) is a (commutative, idempotent) monoid, with $\emptyset$ as identity.

Limitations. While a CFG specifies a syntax, it does not supply an algorithm for parsing. Moreover, it specifies *only* syntax; a programming language also needs *semantics*. Section 3.3 covers one approach to parsing a monoidally-extensible syntax. Section 3.4 describes how Moxy permits extensible semantics.

2.2 Macro-expansion

Consider a simple lisp-like syntax:

s-expressions	e	$::=$	$a \mid (es)$
s-expression lists	es	$::=$	$\mid e\; es$
atoms	a	$::=$	$s \mid n$
numerals	n	$::=$	$0 \mid 1 \mid \ldots$
symbols	s	$::=$	...

We formalize a simplistic version of macro-expansion by defining $\mathsf{expand}(env, e)$, which takes an environment env mapping symbols to their definitions, an s-expression e to expand, and returns ep with all macro invocations recursively replaced by their expansions. In pseudocode:

```
expand($env$, ($s$ $es$)) if $s$ in $env$ = expand($env[s](es)$)
expand($env$, ($es$)) = map(expand($env$, _), ($es$))
expand($env$, $a$) = $a$
```

(We write $env[s]$ for the macro-definition of s in env, which we take to be a function from an argument-list es to its immediate expansion under that macro.)

Where's the monoid? If macros yield a form of extensibility, how do they fit into our monoidal framework? First, we must find the set of possible extensions. Intuitively, *macros* are the extensions we are concerned with. Perhaps the set of all macro-definitions? However, there is no obvious operator to "merge" two macro-definitions.

This suggests that our notion of extension is not powerful enough. So instead of single macros, we take our extensions to be *environments* mapping macro-names to their definitions, like the env argument to expand. Our binary operator $(\cdot)$ merges two environments, so that:

$$\begin{aligned} s \textbf{ in } (env_1 \cdot env_2) \quad &\textbf{iff} \quad (s \textbf{ in } env_1) \vee (s \textbf{ in } env_2) \\ (env_1 \cdot env_2)[s] \quad &= \quad \begin{cases} env_1[s] & \textbf{if } s \textbf{ in } env_1 \\ env_2[s] & \textbf{otherwise} \end{cases} \end{aligned}$$

By convention, $(\cdot)$ is left-biased: if the same symbol is defined in both arguments, it uses the left one.

Finally, note that $(\cdot)$ is associative, and forms an identity with the empty environment; that is, $(\cdot)$ is a monoid.

Monoid-parameterized functions. Observe that expand is a function parameterized by a value of the monoid representing extensions. It demonstrates one way of giving *semantics* to a notion of extensibility: interpret extensions into functions. This is a pattern we will see again in Moxy's parse and compile phases.

2.3 Open recursion and mixins

Open recursion is the problem of, first, defining a set of mutually-recursive functions *by parts*; that is, permitting the programmer to define only some of the functions, and complete the set by defining the rest later; and, second, of allowing the behavior of these functions to be *overridden or extended.*

For example, consider the functions $\mathsf{even}(n)$ and $\mathsf{odd}(n)$, of type $\mathbb{N} \to \mathsf{Bool}$, defined by:

```
even(0) = true
even($n$) = odd($n - 1$)
odd(0) = false
odd($n$) = even($n - 1$)
```

Definition *by parts* means that we could define even and odd separately, as *mixins*, and later combine them into an *instance* that implements both:

```
mixin EvenMixin where
  inherit odd
  even(0) = true
  even($n$) = self.odd($n - 1$)
end
mixin OddMixin where
  inherit even
  odd(0) = false
  odd($n$) = self.even($n - 1$)
end
```

```
instance EvenOdd where
  use EvenMixin
  use OddMixin
end
EvenOdd.even(10)   — returns true
```

Overriding means that a mixin can extend the behavior of a function in a fashion similar to traditional object-oriented subclassing:

```
mixin EvenSpyMixin where
  inherit odd
  even(n) =
    print(n);
    super.even(n)
end
instance EvenOddDebug where
  use EvenMixin
  use OddMixin
  use EvenSpyMixin
end
— prints 2, then prints 0, then returns true
EvenOddDebug.even(2)
```

2.3.1 Semantics of mutual recursion.

Setting aside for a moment the problem of open recursion, consider ordinary, "closed", mutual recursion. Suppose we wish to implement some signature $\Sigma = \langle N, \tau \rangle$ of mutually recursive functions, where N is a set of names and the function $\tau : N \to \mathsf{type}$ assigns each name a type. An *implementation* of Σ is a function giving to each name n a value of type $\tau(n)$. We write Impl_Σ for the type of implementations of the signature Σ:

$$\mathsf{Impl}_{\langle N,\tau\rangle} = \Pi(n : N).\ \tau(n)$$

We represent a group of mutually-recursive function *definitions* as a function from implementations to implementations: we take a self object, and to recursively call the function n, we call $\mathsf{self}(n)$. In this manner we make self-reference explicit. We write Defn_Σ for the type of mutually-recursive function definitions represented this way:

$$\mathsf{Defn}_\Sigma = \mathsf{Impl}_\Sigma \to \mathsf{Impl}_\Sigma$$

For example, our original mutually-recursive definition of even and odd is represented by:

$$\begin{array}{lcl} \Sigma & = & \langle N, \tau \rangle \\ N & = & \{\mathsf{even}, \mathsf{odd}\} \\ \tau(\mathsf{even}) & = & \mathbb{N} \to \mathsf{Bool} \\ \tau(\mathsf{odd}) & = & \mathbb{N} \to \mathsf{Bool} \\ \mathsf{EvenOdd} & : & \mathsf{Defn}_\Sigma \\ \mathsf{EvenOdd}(self)(\mathsf{even})(0) & = & \mathsf{true} \\ \mathsf{EvenOdd}(self)(\mathsf{even})(n) & = & self(\mathsf{odd})(n-1) \\ \mathsf{EvenOdd}(self)(\mathsf{odd})(0) & = & \mathsf{false} \\ \mathsf{EvenOdd}(self)(\mathsf{odd})(n) & = & self(\mathsf{even})(n-1) \end{array}$$

To obtain the desired implementations of even and odd, we take the least-fixed-point of the EvenOdd function:

$$\begin{array}{lcl} \mathsf{even} & = & fix(\mathsf{EvenOdd})(\mathsf{even}) \\ \mathsf{odd} & = & fix(\mathsf{EvenOdd})(\mathsf{odd}) \end{array}$$

For some function $fix : (\alpha \to \alpha) \to \alpha$ satisfying $fix(f) = f(fix(f))$ for appropriate α (here, $\alpha = \mathsf{Impl}_\Sigma$).

2.3.2 Mixins as a monoid

Let's apply our monoid methodology to the problem of semantics for open recursion. We'll tackle definition by parts first, and later expand our semantics to cover overriding as well.

First, we must determine our set of extensions. The extensions we are concerned with are "mixins": partial definitions of a set of mutually recursive functions. The definitions in a mixin have access to a self object, allowing mutual- or self-recursion.

At first glance, this seems very similar to the way we formalized closed sets of mutually-recursive definitions. The only difference is that mixins are permitted to be *partial*, and omit a definition for a particular function in the signature.

However, we can use the same type Defn_Σ to represent mixins if for names n which the mixin M does not define, we simply let $M(self)(n) = self(n)$, passing the undefined method through unchanged. Thus a mixin is represented by a function from implementations to implementations: it takes a self object, and returns that object updated with each of the functions the mixin implements.

For example, EvenMixin and OddMixin are represented by the following functions:

$$\begin{array}{lcl} \mathsf{EvenMixin}(self)(\mathsf{even})(0) & = & \mathsf{true} \\ \mathsf{EvenMixin}(self)(\mathsf{even})(n) & = & self(\mathsf{odd})(n-1) \\ \mathsf{EvenMixin}(self)(\mathsf{odd}) & = & self(\mathsf{odd}) \\ \\ \mathsf{OddMixin}(self)(\mathsf{odd})(0) & = & \mathsf{false} \\ \mathsf{OddMixin}(self)(\mathsf{odd})(n) & = & self(\mathsf{even})(n-1) \\ \mathsf{OddMixin}(self)(\mathsf{even}) & = & self(\mathsf{even}) \end{array}$$

At first, this representation seems pointless, like trying to fit a square peg (mixins) into a round hole (Defn_Σ). After all, if we take $fix(\mathsf{OddMixin})(\mathsf{odd})$, the resulting function diverges for all nonzero inputs!

However, observe that

$$(\mathsf{EvenMixin} \circ \mathsf{OddMixin}) = \mathsf{EvenOdd}$$

(as can be verified by some tedious calculations)!

That is, that we can "combine" mixins by composing them as functions; our monoidal operator is $\circ$. Our identity is just the identity function at type Impl_Σ. Function composition is associative, so we are done.

2.3.3 Mixins with overriding

$$\begin{array}{rcl} \mathsf{Mixin}_\Sigma & = & \mathsf{Impl}_\Sigma \to \mathsf{Impl}_\Sigma \to \mathsf{Impl}_\Sigma \\ \varepsilon(self)(super) & = & super \\ (f \cdot g)(self)(super) & = & f(self)(g(self)(super)) \end{array}$$

$$\begin{array}{lcl} \mathsf{EvenMixin}(self)(super)(\mathsf{even})(0) & = & \mathsf{true} \\ \mathsf{EvenMixin}(self)(super)(\mathsf{even})(n) & = & self(\mathsf{odd})(n-1) \\ \mathsf{EvenMixin}(self)(super)(\mathsf{odd}) & = & super(\mathsf{odd}) \\ \mathsf{OddMixin}(self)(super)(\mathsf{odd})(0) & = & \mathsf{false} \\ \mathsf{OddMixin}(self)(super)(\mathsf{odd})(n) & = & self(\mathsf{even})(n-1) \\ \mathsf{OddMixin}(self)(super)(\mathsf{even}) & = & super(\mathsf{even}) \end{array}$$

$$\begin{array}{lcl} \mathsf{EvenSpyMixin}(self)(super)(\mathsf{odd}) & = & super(\mathsf{odd}) \\ \mathsf{EvenSpyMixin}(self)(super)(\mathsf{even})(n) & = & \mathbf{print}(n); \\ & & super(\mathsf{even}(n)) \end{array}$$

To actually produce an implementation:

$$\begin{array}{rcl} impl & : & \mathsf{Mixin}_\Sigma \to \mathsf{Impl}_\Sigma \\ impl(f) & = & fix(\lambda s.f(s)(\bot)) \end{array}$$

2.4 Web framework middleware

We observe that middleware in many web frameworks (for example, Django) is essentially a stack of pairs of functions, $(\mathsf{Request} \to \mathsf{Request}) \times (\mathsf{Response} \to \mathsf{Response})$, which are composed together via the monoid:

$$\begin{array}{rcl} \varepsilon & = & \langle \mathsf{id}_{\mathsf{Request}}, \mathsf{id}_{\mathsf{Response}} \rangle \\ \langle f_{\mathsf{req}}, f_{\mathsf{resp}} \rangle \cdot \langle g_{\mathsf{req}}, g_{\mathsf{resp}} \rangle & = & \langle f_{\mathsf{req}} \circ g_{\mathsf{req}}, g_{\mathsf{resp}} \circ f_{\mathsf{resp}} \rangle \end{array}$$

top-level	t	$::=$	$d \mid$ **module** $N\ \{t^*\}$
expressions	e	$::=$	$Pn \mid PN \mid l \mid$ `(`e`)` $\mid e \oplus e$
		$\mid$	`\(`$p, ...$`)` $e \mid e$`(`$e, ...$`)`
		$\mid$	**let** d^* **in** e
		$\mid$	**case** e [`|` p `->` e]*
declarations	d	$::=$	**val** p `=` e
		$\mid$	**fun** n`(`$p, ...$`)` `=` e [`|` n`(`$p, ...$`)` `=` e]*
		$\mid$	**tag** N[`(`$n, ...$`)`]
		$\mid$	**rec** d [**and** d]*
		$\mid$	**open** PX
patterns	p	$::=$	$x \mid l \mid PX$[`(`$p, ...$`)`]
operators	$\oplus$	$::=$	`+` $\mid$ `-` $\mid$ `*` $\mid$ `/` $\mid$ `==` $\mid$ `<=` $\mid$ `>=` $\mid$ `<` $\mid$ `>` $\mid$ `;`
		$\mid$	...
module paths	P	$::=$	[N`.`]*
capital names	N		
names	n		
literals	l		

Figure 1. Grammar of Moxy, sans extensions

And then composed onto a base handler function h : Request $\rightarrow$ Response:

$$\mathsf{withMiddleware}(\langle m_{\mathsf{in}}, m_{\mathsf{out}}\rangle, h) = m_{\mathsf{out}} \circ h \circ m_{\mathsf{in}}$$

Middleware "in the wild" is more complicated, having to deal with details such as error-handling, the possibility of early exit, and so forth, but still forms a monoid. The fact that it forms a monoid can even be observed by the way one specifies what middleware to use: via a list, i.e. an element of the free monoid.

2.5 Monad transformers

Broadly speaking, monads as they appear in Haskell represent *effects* which a computation has access to: for example, Maybe represents the possibility of failure; State s represents access to a single mutable location of type s; List represents backtracking nondeterminism.

It is often useful to have multiple kinds of effect; for example, failure *and* state. For this Haskell uses *monad transformers*, type-level functions from monads to monads. Letting **Hask** be the category of Haskell types, a monad such as State s has kind **Hask** $\rightarrow$ **Hask**; a monad transformer such as StateT s has kind (**Hask** $\rightarrow$ **Hask**) $\rightarrow$ (**Hask** $\rightarrow$ **Hask**).

To combine multiple effects, one creates a stack of monad transformers, terminating it with the identity monad Identity. To combine failure (Maybe) with binary state (State Bool) and logging (Writer [String]), we write:

$$\mathsf{WriterT\ [String]\ (StateT\ Bool\ (MaybeT\ Identity))}$$

Clearly monad transformers represent a notion of extensibility, namely "adding effects" to a pure base language. Unsurprisingly, they form a monoid; the operator is simply composition, and the identity is the monad transformer IdentityT. The above can be rewritten (assuming a type-level composition operator $\circ$) as:

$$\mathsf{(WriterT\ [String]} \circ \mathsf{StateT\ Bool} \circ \mathsf{MaybeT)\ Identity}$$

3. Moxy

3.1 A brief introduction to Moxy, sans extensions

Moxy exists to test the hypothesis that monoids are a good way of structuring extensibility. In other respects it is a banal, humdrum language.

The basic grammar of Moxy is given in figure 1. "$e, ...$" indicates a comma-separated list of expressions e; similarly for patterns, "$p, ...$".

Syntactically, Moxy is moderately MLish; semantically, Moxy is somewhat Schemelike. Evaluation is eager; functions take multiple arguments and return one value; recursion is the only looping construct; pattern-matching is the only conditional construct (**if** is implemented as a syntax extension). Moxy has a simple module system that only handles namespacing (no ML-style functors). Strings and numbers are built-in; booleans are defined in the implicitly-imported prelude.

As an example, here is a naïve recursive implementation of the fibonacci function in Moxy:

```
fun fib(0) = 1
  | fib(1) = 1
  | fib(n) = fib(n-1) + fib(n-2)
```

Case is syntactically significant in Moxy: capitalized names are used for modules, tags, and extension points, uncapitalized for everything else.

3.2 Moxy's approach to extensibility: Extension points

Moxy aims to be *meta-extensible*: to permit extensions (for example, a DSL for parsers) that are themselves extensible (for example, defining $a+$ as an abbreviation for $aa*$). It therefore seems difficult to specify in advance what the monoid representing Moxy extensions should be.

Moxy's solution is to let the programmer define their own extension monoids, which Moxy calls *extension points*. To define an extension point, the programmer gives it a name (a Moxy identifier), a value representing the identity element, and a function representing the monoid operator. For example:

```
extension ExtCounts(0, \(x,y) x+y)
```

This defines an extension point named ExtCounts with the monoid $\langle \mathbb{N}, +, 0\rangle$. As Moxy is dynamically typed, the fact that the intended domain of ExtCounts is $\mathbb{N}$ exists only in the programmer's mind.[2]

Having defined an extension point, we can extend it with an **extend** declaration:

```
extend ExtCounts with 2 + 2
```

Within the scope of this declaration, ExtCounts will have a value 4 higher than it otherwise would. Of course, this is quite useless without some way to observe the value of an extension point and use it in parsing.

For this purpose, Moxy comes with built-in extension points which allow extending Moxy's built-in syntax classes: expressions, declarations, and patterns. For example, the InfixExprs extension point permits adding new infix operators to the expression language:

```
extend InfixExprs with
  { TSYM("."): { precedence = 10,
                 parse = [...omitted...] } }
```

3.3 Parsing Moxy

Moxy uses monadic parser-combinators after the style of Parsec to implement a Pratt-style recursive descent parser. In practice what the latter means is that infix operators are handled by means of a table mapping the token for a given operator (say, + for addition) to its precedence, paired with a parser for its right-hand-side. When parsing an expression we keep track of the precedence we are currently parsing at, and if we encounter an operator of looser

[2] Indeed, there is nothing to say that the intended domain is not $\mathbb{Z}$ instead.

precedence we stop parsing and return. Otherwise we recursively invoke the parser for the operator we found, with the expression we've parsed so far as an additional argument.

This permits arbitrary infix, suffix, and mixfix operators. However, it requires that each infix operator have a unique token identifying it; no overloading is possible (at least, not in the parser).

Moxy currently has a separate tokenization phase, which unfortunately limits the expressiveness of parse extensions. We believe this can be eliminated in future by means of a scannerless-parsing technique, such as that used by Parsec's Text.Parsec.Token module.

The monad which Moxy uses for parsing, in addition to behaving like Parsec, also acts as a Reader ParseEnv, where ParseEnv is a datastructure representing all extensions to all extension points currently in scope. A ParseEnv is effectively a mapping from extension points to their values. It is the parameterization of the parser by this value that lets extensions affect parsing (just as the parameterization of expand by a macro-environment lets macro-definitions affect macro-expansion).

For example, when the parser tries to parse the start of an expression, it first examines the current value of the Exprs extension point. Exprs' domain is dictionaries mapping tokens to parsers; its monoid operation is right-biased merge and its identity is the empty dictionary. If the next token is present in the value of Exprs, then we invoke the parser it is bound to. Otherwise we try built-in parse productions such as literals, variables, and function application.

Similar techniques are used for extending declarations and patterns.

3.4 Compiling Moxy

Nodes in Moxy's AST are instances of abstract interfaces. For example, an expression is merely something which *knows how to compile itself*. That is, expressions are "records"[3] with a functional compile field. This compile function, when supplied with a *resolve environment*, returns the intermediate-representation[4] compilation of the expression in question.

The *resolve environment* is effectively merely the lexical environment of the expression, a dictionary which tells the compiler which variable names are in scope and how references to them are to be compiled.

4. Prior work

Lisp and Scheme are obvious predecessors in the search for extensible forms of programming. Moxy is in large part an attempt to replicate the ease and power of Lisp-family macros in a non-Lisp setting.

SugarJ does almost everything Moxy does and more [1]. However, it weighs in at 26k LOC, while Moxy is a mere 2k LOC. Moxy can use extensions per-scope, not just per-file. Moxy has a REPL, but SugarJ's approach is probably compatible with a REPL as well. Moxy allows for non-syntactic notions of scoped extensibility (but I have no motivating examples). The Moxy approach currently has no story for integrating with existing languages (but it probably could be done).

Moxy's use of a Pratt-style parser to aid extensibility is similar to that of Magpie. [4]

Other previous syntactically extensible systems include OMeta [3], Sweet.js, Seed7, and Omar et al's Type-Specific Languages [2].

[3] In this case, represented as hash-tables.

[4] In this case, s-expressions representing Racket code.

References

[1] **SugarJ: Library-based Syntactic Language Extensibility**. Sebastian Erdweg, Tillman Rendel, Christian Kästner and Klaus Ostermann. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 391-406. ACM, 2011.

[2] **Safely Composable Type-Specific Languages**. C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin and J. Aldrich. *European Conference on Object-Oriented Programming (ECOOP 2014) Uppsala, Sweden, July 28 – August 1, 2014*.

[3] **Experimenting with Programming Languages**. Alessandro Warth. Technical Report TR-2008-003, Viewpoints Research Institute, 2009.

[4] **Extending Syntax from Within a Language**. Bob Nystrom. `http://journal.stuffwithstuff.com/2011/02/13/extending-syntax-from-within-a-language/`, 2011-02-13, accessed 2014-09-25.

[5] **Parsec: Direct Style Monadic Parser Combinators For The Real World**. Daan Leijen and Erik Meijer. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.