# From Macros to Reusable Generative Programming*

Shriram Krishnamurthi[1], Matthias Felleisen[1], and Bruce F. Duba[2]

[1] Department of Computer Science
Rice University
Houston, TX 77005-1892, USA
[2] Department of Computer Science
Seattle University
Seattle, WA 98122-4460, USA

**Abstract.** Generative programming is widely used both to develop new programming languages and to extend existing ones with domain-specific sub-languages. This paper describes McMicMac, a framework for generative programming. McMicMac uses tree-transforming macros as language specifications, and enhances them with inherited and synthesized attributes. The enhanced transformers can describe general compilation tasks. Families of these specifications are grouped into mixin-like collections called vocabularies. Programmers can define new languages by composing these vocabularies. We have implemented McMicMac for Scheme and used it to build several systems, including the DrScheme programming environment. The principles of McMicMac carry over to other languages and environments.

## 1 Introduction

Generative programming is an old and good idea—write programs to write your programs for you. This idea has been applied widely, from compilation to creating domain-specific languages to the mass-production of components. It is gaining increasing prominence due to its potential economic impact, because it can both reduce manual labor and increase the efficiency and correctness of programs.

In this paper, we describe a general framework for generative programming in Scheme [21]. Concretely, we describe McMicMac, a framework developed at Rice University. McMicMac supports the production of extensible language components. A programmer can generate new languages by composing language components. Because the components are parameterized over the base language, McMicMac programmers can reuse the same language-defining components to build many different languages. The system thus greatly simplifies the design and implementation of domain-specific languages as extensions of base languages.

Our system plays a crucial role in the construction of a large programming environment, DrScheme [12]. Conversely, the process of building DrScheme has

* This work is partially supported by NSF grants CCR-9619756, CDA-9713032, and CCR-9708957, and a Texas ATP grant.

helped us debug and refine the design of our system. McMicMac has also been used to build tools for languages other than Scheme. Starting with a type-checker and compiler for a parenthetical version of Java, it has been used to create language extensions representing design patterns [16] and other specifications [26].

The rest of this paper is organized as follows. Section 2 provides an extended example that illustrates some of the kinds of abstractions that generative programming makes possible. Section 3 describes the structure of McMicMac through a series of examples, and section 3.3 and section 4 refine this discussion. Section 5 summarizes the deployment of McMicMac. Section 6 discusses related efforts, and section 7 offers concluding remarks and suggests directions for future work.

## 2 An Illustrative Example

Suppose a programmer wanted to distribute a library that creates and executes finite-state automata. The library of types and procedures (or classes with methods) should hide the actual representation used for the automata, because there are at least two different automata representations: data structures and executable values. Data structures are useful for manipulating automata as data, e.g., for minimizing automata. They can also be run via an interpreter, though this can be inefficient. If the automata are being created solely for execution, then it is typically much more efficient to translate them directly into code, such as functions or labeled statements.

Hence, the library should offer programmers a way to specify in-line automata in a representation-independent, yet intuitive, manner. This would allow clients to write expressions such as that shown in figure 1. The example uses a new construct, **automaton**, to define an automaton that, starting in *1-state*, checks whether a stream of 0s and 1s begins with 0 and then alternates strictly between the two values.

The specification in figure 1 might translate into the Scheme code shown in figure 2. It represents automata as procedures that consume an input stream. The states are nested, mutually-recursive procedures. Each procedure represents one state's transition relation. All unexpected inputs generate an exception, *unexpected-input-exn*, so the empty transition relation represents an error state. If the automaton attempts to inspect past the end of a finite input stream, a different exception is raised, which a client can handle to observe successful completion. Because programming languages like Scheme optimize calls in tail position to jumps or "goto"s, the state transitions in the example are quick and accumulate no evaluation context (colloquially, "stack space") [7].

Unfortunately, the library programmer cannot define **automaton**, because it is not a procedure. The sub-terms of **automaton** are not expressions in Scheme, as procedure arguments must be; rather they are terms in a distinct, domain-specific language. Furthermore, **automaton** is a *binding* construct (as clarified by figure 2), which cannot be defined procedurally. Therefore, **automaton** cannot be defined in most traditional programming languages.

```
(automaton 1-state
  (0-state ((0 ⟶ error-state)
            (1 ⟶ 1-state))
  (1-state ((0 ⟶ 0-state)
            (1 ⟶ error-state)))
  (error-state)))
```

**Fig. 1.** An Automaton Description

```
(lambda (input-stream)
  (letrec
      ((0-state (lambda ()
                  (case (next-token input-stream)
                    ((0) (error-state))
                    ((1) (1-state))
                    (else (raise (make-object unexpected-input-exn))))))
       (1-state (lambda ()
                  (case (next-token input-stream)
                    ((0) (0-state))
                    ((1) (error-state))
                    (else (raise (make-object unexpected-input-exn))))))
       (error-state (lambda ()
                      (case (next-token input-stream)
                        (else (raise (make-object unexpected-input-exn)))))))
    (1-state)))
```

**Fig. 2.** Compiled Automaton Representation

The automaton library might contain other operations of this form. For instance, the library may provide a means for interleaving the execution of two automata. This would enable the client programmer to write

**(run/alternating** (*M1 stream-1*) (*M2 stream-2*))

which runs automaton *M1* on stream *stream-1* and *M2* on *stream-2* in strict alternation. In this case, even though both sub-terms of **run/alternating** are legal Scheme expressions, Scheme's call-by-value evaluation order would first run *M1* on *stream-1* until termination—which may never occur—before it begins to run *M2*. Thus **run/alternating** also cannot be a procedure in a call-by-value language.

Both constructs illustrate useful and important abstractions that help programmers write software effectively. These abstractions are not *procedural*, however. Instead they define notations that are not part of the language's syntax. In other cases, they require behavior that is different from what the language's semantics specifies. More generally, **automaton** and **run/alternating** are both *linguistic* abstractions, i.e., they create a little language within a larger language

```
(define-macro (automaton ⟶)
  (automaton start-state
             (state-name (input ⟶ new-state) ...) ...)
  ⟹ (syntax
       (lambda (input-stream)
         (letrec ((state-name
                   (lambda ()
                     (case (remove-token input-stream)
                       ((input) (new-state)) ...
                       (else (raise (make-object unexpected-input-exn)))))) ...)
           (start-state)))))
```

**Fig. 3.** Automaton Macro

to accomplish some specialized task. Generative programming frameworks must support such language construction.

## 3 The McMicMac Framework

McMicMac is a framework for creating languages such as that for automata. We explain McMicMac through a series of examples reflecting increasingly complex protocols. The examples are intentionally simplistic in flavor, but they correspond to some of the non-trivial uses we have encountered while building DrScheme.

### 3.1 Macros

Examples like **automaton** and **run/alternating** are expressible as McMicMac *macros*. The macros of McMicMac are descendants of those in Lisp and Scheme [21,33]. They transform tree-shaped data, rather than manipulating flat data like the string-processing macros in the C pre-processor [22]. In short, macros implement a simple form of extensible parsing.

A parser is conceptually a table of rules that map syntactic shapes to code. When the input matches a shape, called a *trigger*, the parser looks up the trigger's transformation rule, called the *elaborator*, and uses it to produce abstract syntax. The parsing table is traditionally fixed, thus limiting the input language a parser can recognize. Macro definitions add rules to a parser's table. Unlike traditional parse rules, though, macro rules do not directly generate abstract syntax. Instead, they generate terms in the source language. The parser then re-analyzes the generated term and continues this process until it obtains a canonical form.

In McMicMac, the programmer defines triggers using a pattern-matching notation originally due to Kohlbecker and Wand [24]. When an input term matches

```
(define-macro (automaton ⟶)
  (automaton start-state
             (state-name (input ⟶ new-state) ...) ...)
  ⟹ (syntax
      (make-automaton-rep start-state
        (list (make-state-rep state-name
                (make-transition-rep input new-state) ...)
              ...))))
```

**Fig. 4.** Alternate Automaton Macro

a trigger, the matcher generates a pattern environment that maps pattern variables to the corresponding source terms in the input. It then invokes the elaborator to generate a source term. This term can be parameterized over sub-terms in the input. The elaborator extracts these input sub-terms from the pattern environment.

Figure 3 presents a concrete example: the macro for the **automaton** construct of section 2. The keyword **define-macro** is followed by a set of literals (here, **automaton** and ⟶) that may appear in the input. The literals are followed by the trigger. All symbols in the trigger that do not appear in the literal set are *pattern variables*. A pattern followed by ellipses (...) matches zero or more instances of the pattern. It binds each pattern variable to the sequence of sub-terms that correspond to the pattern variable's position in the matching instances. Ellipses can be nested arbitrarily deep.

The macro definition specifies an elaborator following the ⟹ keyword. The elaborator uses **syntax** to construct a new source term. The **syntax** form consumes a template and converts the template into a term in the source language by replacing all pattern variables with their bindings from the (implicit) pattern environment. Thus, in the output term, *start-state*, *state-name*, *input* and *new-state* are replaced with the corresponding source text in the input expression, while all other names are inserted literally.[1]

Macros of this sort have traditionally been put to four main uses:

- to affect the order of evaluation. For instance, **run/alternating** requires delayed evaluation in a call-by-value language. The macro can wrap the expressions in procedures that are invoked to control stepping.
- to create new binding constructs. The procedural translation of **automaton** turns the state-names into binding and bound occurrences of variables.
- to mask the creation of a data-structure. It is straightforward to implement an **automaton** macro that produces a traditional data structure representation of automata.
- to represent structural program properties, such as uses of design patterns.

[1] We have elided from our specification many McMicMac features that are useful in practice; e.g., the macro writer can specify guards on the structure of sub-terms.

```
(define-micro (if)                 ;; for if's
  (if test then else) ⟹ (lambda ()
                          (make-if-IR
                            ((dispatch (syntax test)))
                            ((dispatch (syntax then)))
                            ((dispatch (syntax else))))))
```

**Fig. 5.** A Micro Specification

As the **automaton** example illustrates, a single macro can serve several of these purposes simultaneously. This is especially likely to happen in the case of *data languages*, whose constructs are supposed to mask their representation of the data from the client programmer, since the same source can be used to generate either representation. Figure 4 presents an alternate compiled representation for automata: instead of creating procedures, it generates a data structure.

Different applications can choose alternate expansions (using the mechanism described in section 3.3) without any intervention from the user who specifies the automata. Linguistic extensions are especially important in this context, because they are often the only way to mask these concrete representations. In most languages, for instance, it is impossible to define a construct that elaborates into a procedure. This makes it extremely difficult, if not impossible, to write the **automaton** abstraction through any other means.

## 3.2 Beyond Conventional Macros

Though conventional macros can describe many interesting linguistic abstractions, they are not powerful enough for many other generative-programming tasks. McMicMac therefore generalizes macros in several ways. These generalized macros propagate information about the program to guide elaboration.

**From Expansion to Parsing** Parsers have the type scheme

$$source \longrightarrow \text{IR}$$

where *source* is the type of source expressions and IR that of the intermediate representation. In many cases, McMicMac programmers writing language extensions need the power to generate terms of type IR directly. McMicMac thus allows programmers to create such elaborators, called *micros*.

In principle, micros have the type

$$source \longrightarrow \text{IR} \ ,$$

in contrast to macros, which denote source-to-source rewriting functions:

$$source \longrightarrow source \ .$$

```
(define-micro (if)              ;; for if's
  (if test then else) ⟹ (lambda (env)
                           (make-if-IR
                             ((dispatch (syntax test)) env)
                             ((dispatch (syntax then)) env)
                             ((dispatch (syntax else)) env))))

(define-micro (lambda)          ;; for lambda's
  (lambda vars body) ⟹ (lambda (env)
                           (make-lambda-IR vars
                             ((dispatch (syntax body)) (append vars env)))))
```

**Fig. 6.** Micros with Attributes

In practice, the parsing process described in section 3.1 has two parts: the elaborators that create IR and the dispatcher that does pattern matching and invokes an elaborator. The dispatcher, called *dispatch*, has type

$$\mathit{source} \longrightarrow \mathit{micro}\ ,$$

i.e., given a source term, it returns the corresponding micro. Micros are elaborators represented as procedures of no arguments that create IR, whose type we denote as

$$() \longrightarrow \text{IR}\ .$$

(The reason for this seemingly needless level of indirection will become clear in the following sections.) The output of micros, unlike that of macros, is not automatically expanded again. If it were automatically expanded, this could result in a type conflict. Therefore, a micro must invoke **dispatch** to reduce source terms to IR.

Figure 5 shows the micro definition for a simple conditional construct. Like **define-macro**, **define-micro** is followed by a list of literals and a trigger pattern. To the right of the ⟹ keyword is the specification of the micro's elaborator, a procedure of no arguments. The elaborator uses *make-if*-IR to construct the IR representation of **if** expressions. The invocation

((**dispatch** (**syntax** *test*)))

extracts the source term corresponding to *test* from the pattern environment, uses *dispatch* to obtain the corresponding micro, which is a procedure of no arguments, and invokes the micro to generate the IR value for *test*.

**Attributes** Suppose a programmer wants a simple value inspection facility. Specifically, the expression (**dump**) should print the names and values of all the variables bound in the lexical scope. Provided we have access to the names of all the variables in that lexical context, the transformation associated with (**dump**) is quite straightforward. McMicMac allows programmers to make this contextual information explicit by associating *attributes* with the dispatcher. Thus a programmer can declare the type of a micro to be

$$env \longrightarrow \textsc{ir}$$

where *env* is the type of the lexical environment. The micro can inspect this environment to determine the names of the bound variables.

This type generalizes to

$$attr \cdots \longrightarrow \textsc{ir}$$

to indicate that there can be several attributes. Every micro must accept all the attributes, and must propagate them to micro dispatches on sub-terms. Figure 6 presents the definition for **lambda** (which affects the set of lexical variables listed in *env*) and a revised definition of **if** (which doesn't).

To use McMicMac, an application must invoke **dispatch** on the source program while supplying appropriate values for all the attributes. The result of invoking **dispatch** is an IR value, which the program can use for subsequent processing. Some applications use the same type for the source and IR, i.e., they only exploit attributes, not the ability to transform representations.

**Threaded Attributes** One possible IR to choose may be the set of values in the language. In that case, the "parser" may convert programs to their final answers, i.e., it may really be an interpreter. We expect McMicMac to deal with such transformations too. They are useful for prototyping small embedded domain-specific languages, or for optimizing code-generators. Attributes can represent various aspects of the language's evaluation. In figure 6, for example, the environment maintains only a list of names bound in each context, but in an interpreter, the environment could map names to locations or values.

Non-trivial languages, though, have two kinds of attributes. Some attributes, e.g., environments, are *functional*, meaning they do not represent computational effects. Other attributes, however, are *threaded*. They are affected in the processing of sub-terms, and their order of propagation from the processing of one sub-term to another matters. A canonical example of such an attribute is the store. If the same store were passed to all sub-terms, then side-effects in the evaluation of one would not be visible in the other. Micros therefore return the updated values of threaded attributes along with the IR. Thus the type of a micro in the interpreter implementation can be

$$env \times store \longrightarrow \textsc{ir} \times store$$

or, in general, micros can have a type with the shape

$$funattr \cdots \times threadattr \cdots \longrightarrow \textsc{ir} \times threadattr \cdots \ .$$

Once again, it is the micro programmer's responsibility to invoke McMicMac, provide arguments for the attributes, accept the final IR value and the values of the threaded attributes, and process them.

```
(define-micro (set!)                ;; for set!'s
  (set! var val) ⟹ (lambda (env store)
                       (let/values ((val-value val-store)
                                    ((dispatch val) env store))
                         (values ;; the value:
                                 (void-value)
                                 ;; the new store:
                                 (extend store var val-value)))))

(define-micro ()              ;; for function applications—no literals
  (fun arg) ⟹ (lambda (env store)
                   (let/values ((fun-value fun-store)
                                ((dispatch fun) env store))
                     (let/values ((arg-value arg-store)
                                  ((dispatch arg) env fun-store))
                       ;; functions must return value/store pairs
                       (fun-value arg-value arg-store)))))
```

**Fig. 7.** Threaded Attributes

We illustrate this new form of micro with two definitions in figure 7 that implement a stateful language in a purely functional manner using the store-passing style technique from denotational semantics. (**set!** is Scheme's assignment statement.) The code uses Scheme's multiple-value facility to return the actual value and the potentially modified store.

### 3.3 Modular Specifications

We have thusfar discussed the kinds of transformations that programmers can express. In this section, we discuss how programmers can group these transformations into resuable units.

**Vocabularies** Most programming languages consist of several sub-languages: those of expressions, statements, types, argument lists, data, and so on. The programmer must therefore specify which sub-language a micro extends. McMicMac provides *vocabularies* for this purpose. A vocabulary is a grouping of related micros. All micros in a vocabulary must satisfy the same type signature. Figure 8 illustrates the revised declarations from earlier examples. Each micro declares membership in a vocabulary just before specifying its literal set.

The sum of declarations in a vocabulary specifies the syntax and the elaboration rules of a language. Put differently, a vocabulary describes the syntax table that is used by **dispatch**, and must therefore be a parameter to **dispatch**. We update the type of **dispatch** from section 3.2 to reflect this:

$$source \times vocab \longrightarrow micro\ .$$

```
(define scheme-exprs                      (define automata
  (make-vocabulary))                        (make-vocabulary))

(define-micro scheme-exprs (set!)         (define-micro automata
  (set! var val) ⟹                            (automaton ⟶)
    (lambda (this-vocab env store)          (automaton ···) ⟹
      ···))                                   (lambda (this-vocab) ···))

(define-micro scheme-exprs ()             (define-micro automata
  (fun arg) ⟹                                  (run/alternating)
    (lambda (this-vocab env store)          (run/alternating ···) ⟹
      ···))                                   (lambda (this-vocab) ···))
```

**Fig. 8.** Vocabulary Specifications

```
(define compiler (make-vocabulary))       (define analysis
(define-micro compiler (let)                (make-vocabulary))
  (let ((var val) ...) body) ⟹            (define-micro analysis (let) ···)
    ···)                                  (define-micro analysis (letrec) ···)
(define-micro compiler (letrec)
  (letrec ((var val) ...) body) ⟹         (define analysis-language
    ···)                                    (extend-vocabulary scheme-exprs
                                                               analysis))
(define compiler-language
  (extend-vocabulary scheme-exprs
                     compiler))
```

**Fig. 9.** Tool-Dependent Expansions

The change in the type of **dispatch** forces us to update the programming pattern for micros. Each recursive call to **dispatch** must pass along a vocabulary, which the invoked micro must accept.

As we describe below, however, a micro may not always know which vocabulary it is in. Micros therefore take the vocabulary from which they were selected as an argument—in figure 8, each micro accepts a vocabulary, *this-vocab*, as its first argument—which they use to process sub-terms in the same language; alternatively, they can choose a different vocabulary for sub-terms in other languages. For instance, a function declaration may have some sub-terms in the expression language and others in the language of types. Micros therefore have the type scheme

$$vocab \times funattr \cdots \times threadattr \cdots \longrightarrow \text{IR} \times threadattr \cdots \ .$$

**Composing Vocabularies** McMicMac actually allows programmers to build vocabularies by extending and composing them. Thus programmers can divide a language into sets of related features, and compose these features to build a processor for the complete language. A programmer can also create an extension vocabulary that overrides some definitions in a base vocabulary with tool-specific constraints. This explains why a micro may not know which vocabulary it is in; after all, the non-overridden micros of the base are also in the new, composite vocabulary. This scenario is analogous to instance variables in an object-oriented language that reside both in a class and in its extensions.

With vocabularies, we can generate programs in various interesting ways:

– While traditional transformation techniques are limited in where they can be applied—macros and templates are usually restricted to the expression or statement languages—the McMicMac programmer can write transformations for any sub-language. For instance, we have used it to define abbreviations over types and to extend the language of procedural parameters.
– In realistic programming environments, different program-processing tools often have differing views of the underlying language. For example, a compiler might translate the binding construct **let** (which creates non-recursive local bindings) into a local function application while treating **letrec** (which introduces mutually-referential local bindings) as a primitive. In contrast, a polymorphic type inference engine might treat **let** as a core form, while it will transform **letrec** into a more primitive term. These distinctions are easy to express through vocabularies, as shown in figure 9. The function *extend-vocabulary* extends the language of its first argument with the triggers and elaborators of the second, overriding clashes in favor of the second.
– Some languages allow programmers to write lexically-scoped macros [21]. This is easy to define in McMicMac. The micro for a lexical macro construct creates a temporary vocabulary, populates it with the local macro, and extends the current language with the new macro. Because these are local, not global, changes, the language extension disappears when the body has been parsed, so terms outside this lexical context are unaffected.
– A programmer can use vocabularies to organize several traversals over the program. Typically, earlier passes synthesize information for later passes. For example, a programmer may want to add first-class closures to an object-oriented language like Java 1.0. The translator that implements this transformation would need (1) to determine the free variables of the closure's body; (2) to create a class to represent the closure and move its definition to the top-level, as required in many languages; and, (3) to rewrite the creation and uses of the closure.
  A series of vocabularies solves this problem elegantly. The first maintains the lexical environment while traversing code (figure 6); when it encounters an instance of the closure construct, it traverses the body with a vocabulary that computes the set of free variables. This list of free variables is the IR of this vocabulary. It can then generate the class definition. A threaded attribute accumulates top-level definitions created in internal contexts and propagates them outward. Finally, another vocabulary rewrites the creation and use expressions. Determining uses can be done either from type information or through a dynamic check, depending on the target language.

– If a larger language is constructed by composing smaller language layers, programmers can define restricted versions of the larger language by leaving out some layers. We have found this ability especially useful in the DrScheme programming environment [12], which presents the Scheme programming language as a sequence of increasingly complex sub-languages. This hides the complexity of the complete language from the student; in particular, it flags terms that are errors in the linguistic subset but that might be legal—though often not what the student expected—in the full language. This provides much better feedback than an environment for just the complete language would provide, and considerably improves the learning experience.

## 4 Extensibility and Validation

To implement a system like McMicMac in other languages, programmers must resolve the tension between extensibility and validation:

**Extensibility** There are two main sources of extensibility in McMicMac, both related to vocabularies:
  - Vocabularies resemble classes with inheritance; since they have variable parent vocabularies, they are essentially mixins, which enhance reuse by allowing the same class-extensions to be applied to multiple classes.
  - The pattern of passing the vocabulary as an argument accomplishes the effect of the **this** keyword found in many object-oriented languages. It ensures that micros can be reused without having to know about all future extensions. Using this style, programmers can also experiment with interesting language extensions, such as the lexical macros described in section 3.3.

**Validation** Information is communicated through the attributes; the threaded attributes are fundamentally denotational, or monadic [31, 35], in nature. The problems associated with composing vocabularies are thus the same as those with composing arbitrary programming languages.

There are some efforts to design sound type systems for mixins [15], but these are still preliminary. There are unfortunately few type systems that type mixins in their own right and also support the traditional functional types that have been used to represent monadic information. Some of issues that arise from the interplay of types and extensibility are discussed in greater detail by Krishnamurthi, et al. [27].

We have purposely restricted our attention to Scheme, which has no native static type discipline. While the lack of a type system has obvious disadvantages, it has also had its benefits. Because there are no candidate type systems that can cleanly capture the properties we are interested in, restricting ourselves to any one type system would have complicated the design of McMicMac. We were instead interested in first studying the programs that generative programmers write, and can now focus on constructing a type system that supports such programs.

## 5 Implementation Experience

Over the past five years, we have implemented several generations of McMicMac. The current implementation consists of over 10,000 lines of Scheme. It includes a "standard system", which consists of parsing vocabularies for different parts of Rice's version [14] of the Scheme programming language, including the functional core, side-effecting features, modules and signatures, and the class system. The implementation is efficient enough for daily use as the core of a widely-used programming environment, DrScheme [12], and as a pre-processor by various tools including a stepper, static debugger [13] and compiler. We have also exploited McMicMac's attributes to build processors for typed languages, notably for a parenthesized representation of ClassicJava [15].

Our implementation provides source-object correlation [10] and transformation tracking [26], which is extremely useful for interactive programming environments. It eliminates most of the feedback comprehension problems present in traditional macro and template systems. All the tools in DrScheme use this information to provide source-level feedback to the user. This also greatly facilitates linguistic prototyping for features that can be translated, sometimes quite elaborately, into existing ones. The translation ensures that we can reuse existing tools such as the static debugger, and the source correlation hides the complexity of the translation from the user.

## 6 Related Work

The ideas behind generative programming have a long history. Macros appear to have originally been proposed by McIlroy [29], and a symposium in 1969 [5] consolidated progress on extensible languages. Macros have also had a long history of use in various versions of Lisp, and are incorporated into Common Lisp [33], Scheme [21] and others.

C++ has recently added support for a limited form of generative programming called templates. These are intimately connected with the type system of the language, and are thus useful for describing type-based transformations. They have been especially successful at generating families of related components [8].

The Scheme programming language has offered some of the most innovative macro systems. Kohlbecker and others [23–25] introduced both pattern-based rewriting and hygiene for macros. Dybvig, et al. [9] describe a macro system that offers some of the flexibility of micros, but without the structure. Dybvig, et al. [10] present the first source-correlating macro system. Taha and Sheard [34] have designed a macro-like system for ML that lets programmers nest metaprogramming annotations to arbitrary depth, and ensures that the resulting generated code is type-correct. We achieve a similar effect using the static debugger MrSpidey [13] in conjunction with source-correlation.

McMicMac is also related to parser generators such as Yacc [19]. Several researchers have extended these works to adaptable grammars [6]. Some of the most recent work has been done by Cardelli et al. [4], which shows how similar systems can be constructed for non-parenthesized syntaxes. While this system can be used to restrict and extend syntax, it offers no support for organizing elaborator specifications analogous to McMicMac's attributes, and does not present languages as modular layers.

Our extensibility protocol is based on mixins, which are a programming pattern used in Common Lisp [33], and were first formalized by Bracha [3]. The use of mixins in McMicMac lead to a formalized model of mixins for a Java-like language [15]. Our protocol is closely related to that of Smaragdakis and Batory [32], which is a successor to the model of Batory and O'Malley [2]. The work most similar to McMicMac appears to be Batory, et al.'s JTS [1]. JTS shares many common characteristics, including the presentation of languages as layers. The emphases of the two systems appear to be slightly different. JTS is concerned primarily with the construction of component systems, while McMicMac concentrates on the programming forms that simplify specifications.

Some researchers have proposed using languages like Haskell [18] and ML [30] for embedding domain-specific languages, employing higher-order functions, the language's ability to define new infix operators, and (in Haskell's case) laziness to define new notations [11, 17]. These features are, however, not enough to construct abstractions like **automaton** (figure 1). Kamin and Hyatt [20] describe the problem of lifting existing operators to deal with domain-specific constraints; this can also be addressed by generative programming. Finally, Kamin and Hyatt's language reflects a common problem when language extensions cannot define new binding forms. Even though ML already has lexical environments, they must create their own methods and conventions for managing these, thereby inhibiting reuse.

## 7 Conclusions and Future Work

We have described McMicMac, a generative programming system for Scheme. McMicMac extends traditional macro-based generative programming techniques in several ways. First, it lets programmers attach attributes to specifications, which provides a way to propagate information during elaboration, and to guide elaboration itself. Second, it provides a structuring construct called *vocabularies*. Vocabularies are analogous to mixins in object-oriented languages, and permit programmers to define language fragments as components that can be composed. We have used McMicMac to build several tools, including the widely-used DrScheme programming environment [12].

There are two main areas for future work. First, McMicMac is concerned with generative programming in general, and is independent of any notion of components. It has been used in the absence of components, within components, and to generate components. Because linguistic abstractions are different from traditional, procedural abstractions, the integration of components with generative programming can lead to unexpected consequences that should be studied carefully.

Second, the programming style in McMicMac is very much like that used with the Visitor pattern in object-oriented programming [16]. Though we do not show it in this paper, McMicMac allows programmers to separate the specification of triggers from that of the elaborators, so that the elaborator specifications more closely resemble traditional object-oriented programs. Some McMicMac specifications could benefit from higher-level specifications such as adaptive programming [28]. The traditional presentation of adaptive programming, however, does not integrate smoothly with the typed style of specification in McMicMac.

**Acknowledgements**

We thank Robby Findler, Cormac Flanagan and Matthew Flatt for their feedback on McMicMac. We also thank the referees for their comments.

## References

1. Batory, D., B. Lofaso and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *International Conference on Software Reuse*, June 1998.
2. Batory, D. and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.
3. Bracha, G. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, March 1992.
4. Cardelli, L., F. Matthes and M. Abadi. Extensible syntax with lexical scoping. Research Report 121, Digital SRC, 1994.
5. Christensen, C. and C. J. Shaw, editors. *Proceedings of the Extensible Languages Symposium*. Association for Computing Machinery, 1969. Appeared as *SIGPLAN Notices*, 4(8):1–62, August 1969.
6. Christiansen, H. A survey of adaptable grammars. *ACM SIGPLAN Notices*, 25(11):35–44, November 1990.
7. Clinger, W. D. Proper tail recursion and space efficiency. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, June 1998.
8. Czarnecki, K. and U. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, 1999.
9. Dybvig, R. K., D. P. Friedman and C. T. Haynes. Expansion-passing style: A general macro mechanism. *Lisp and Symbolic Computation*, 1(1):53–75, January 1988.
10. Dybvig, R. K., R. Hieb and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.
11. Fairbairn, J. Making form follow function: An exercise in functional programming style. *Software—Practice and Experience*, 17(6):379–386, June 1987.
12. Findler, R. B., C. Flanagan, M. Flatt, S. Krishnamurthi and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs*, pages 369–388, 1997.
13. Flanagan, C., M. Flatt, S. Krishnamurthi, S. Weirich and M. Felleisen. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

14. Flatt, M. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.
15. Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *Symposium on Principles of Programming Languages*, pages 171–183, January 1998.
16. Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Personal Computing Series. Addison-Wesley, Reading, MA, 1995.
17. Hudak, P. Modular domain specific languages and tools. In *International Conference on Software Reuse*, 1998.
18. Hudak, P., S. Peyton Jones and P. Wadler. Report on the programming language Haskell: a non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5), May 1992. Version 1.2.
19. Johnson, S. C. YACC — yet another compiler compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, USA, 1975.
20. Kamin, S. and D. Hyatt. A special-purpose language for picture-drawing. In *USENIX Conference on Domain-Specific Languages*, 1997.
21. Kelsey, R., W. Clinger and J. Rees. Revised$^5$ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), October 1998.
22. Kernighan, B. W. and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, 1988.
23. Kohlbecker, E. E., D. P. Friedman, M. Felleisen and B. F. Duba. Hygienic macro expansion. In *ACM Symposium on Lisp and Functional Programming*, pages 151–161, 1986.
24. Kohlbecker, E. E. and M. Wand. Macros-by-example: Deriving syntactic transformations from their specifications. In *Symposium on Principles of Programming Languages*, pages 77–84, 1987.
25. Kohlbecker Jr, E. E. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, August 1986.
26. Krishnamurthi, S., Y.-D. Erlich and M. Felleisen. Expressing structural properties as language constructs. In *European Symposium on Programming*, March 1999.
27. Krishnamurthi, S., M. Felleisen and D. P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming*, pages 91–113, July 1998.
28. Lieberherr, K. J. *Adaptive Object-Oriented Programming.* PWS Publishing, Boston, MA, USA, 1996.
29. McIlroy, M. D. Macro instruction extensions of compiler languages. *Communications of the ACM*, 3(4):214–220, 1960.
30. Milner, R., M. Tofte and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.
31. Moggi, E. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1990.
32. Smaragdakis, Y. and D. Batory. Implementing layered designs and mixin layers. In *European Conference on Object-Oriented Programming*, pages 550–570, July 1998.
33. Steele, G. L., Jr., editor. *Common Lisp: the Language.* Digital Press, Bedford, MA, second edition, 1990.
34. Taha, W. and T. Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, 1997.
35. Wadler, P. The essence of functional programming. In *Symposium on Principles of Programming Languages*, pages 1–14, January 1992.