# A Modular Interpreter In Scheme With Objects


Steven E. Ganz*
Indiana University

Daniel P. Friedman†
Indiana University



**Abstract**

The problem of writing modular interpreters for functional languages has largely been studied from the perspective of statically typed languages. This paper explores the use of an object-oriented extension of Scheme for this purpose, and finds it to be largely successful. Use of a dynamically typed language has the advantages of greater uniformity and flexibility.


## 1 The Problem of Extensibility, the Test Case of Modular Interpreters and the Search for a Solution in Object-Oriented Programming

Extensibility is the ability of a program to be adapted to new tasks without accessing its source code [7]. Interpreters make an interesting extensibility problem, because in the terminology of Duponcheel [4], there are both syntactic and semantic aspects. By syntax, here, he means abstract syntax, *i.e.*, the logical structure of expressions. By semantics, he means the computations and values engendered by those expressions. This extensibility problem should be contrasted with another that has also been studied extensively, that of adding operations to an object-oriented system [3, 5].

On the syntactic side of our problem, both in the cases of program and of data, the extension must be made *retroactively* to recursions within the original version. Extending a language with a new form does not merely add expressions of that type to the language. It also adds expressions of all preexisting types, with the new form substituted as a subexpression [12]. Similarly, when a new clause is added to an interpreter to handle the new form, this updated interpreter must be used in recursive calls in originally existing clauses.

On the semantic side, computations can be usefully represented using a monad as the result type of the interpeter. When a monadic interpreter is extended, not only interpreter procedures but monadic operators as well must be replaced in original interpreter clauses. Each new layer provides new definitions of return and extend in terms of the old monad, and lifted versions of the other operators. Another semantic issue is the value type. The value type indicated in any module is not intended to fully define the type, only to provide a lower bound. The actual value type may be any supertype of that specified. Finally, the actual computation type may not be the one specified in a module, but one generated from that using a monad transformer. These semantic issues have been described in detail by Liang, Hudak and Jones [9]. Duponcheel emphasizes the syntactic issues. His Gofer implementation exclusively used constrained parametric polymorphism through type classes, and a higher-order version through constructor classes [6]. The approach amounts to always extending non-recursive versions, and then applying a recursion operator to the result of the extension. These issues had earlier been studied by Cartwright and Felleissen [2], who provided a module-based solution, commenting that it is also possible to "use an object-oriented programming language in the spirit of CLOS". More will be said of their approach later.

Why go beyond the success of static type-based solutions to try a dynamic object-based solution? There are several reasons. First, the emphasis on a uniform hierarchical structure in object-oriented programs makes them easier to understand. The solution based on constructor classes uses inheritance directly in the definition of the computational metalanguage, but not in the definition of expressions. The definition of basic expression types use datatype variants; these are combined into a larger language using a Sum datatype. The definition of values are even more obscure, using a subtype typeclass in place of the type-class mechanism itself.

More fundamentally, Lang and Pearlmutter, in presenting their object system, Oaklisp, discuss the language virtue of "uniformity of temporal semantics" [8]. By this, they mean that anything that can be done at compile-time ought to be doable at run-time as well. A uniform temporal semantics leads to a simpler language design, since dependen-


*This work was supported in part by the National Science Foundation under grant CDA-9312614.


Author's address: Department of Computer Science, Indiana University, Bloomington, Indiana 47405. sganz@cs.indiana.edu.



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


Author's address: Department of Computer Science, Indiana University, Bloomington, Indiana 47405. dfried@cs.indiana.edu.

cies of compile-time constructs on run-time data need not be specifically prohibited. For the same reason, it leads to a more flexible and powerful programming environment. One might wish to give up the assumption that the syntax and semantics of a language are known *á priori* and consider situations where elements are provided at run-time by the user. When type information is statically available, it can still be taken advantage of by a good compiler.

Section 2 presents the interface to the object system. Section 3 demonstrates the use of the object system in writing modular interpreters. Section 4 concludes.

## 2 The Object System Interface

The object system we use is similar to Oaklisp [8], which prided itself on providing the full functionality of an object-oriented language while remaining true to Scheme's philosophy. We go even further in both areas. Oaklisp provided multiple inheritance, first-class types and single-dispatch generic functions, along with a metaclass facility. We also provide multiple inheritance and first-class types, along with multiple-dispatch generic functions, discussed in their future work section. We also allow for the functional extension of generic functions, rather than the standard imperative extension.

Another difference is that our system is implemented directly on top of Scheme, rather than through a compiler. Although we do provide metaclasses, we cut back somewhat in not providing MOP support. This can and probably will be added.

The object system was designed with the emphasis on functionality and not efficiency. Multiple inheritance is implemented with explicit, shared subobjects, rather than the more standard implicit subobjects [11, 10]. Slot access is dynamic. Thus, the time for slot access is proportional to the height of the slot definition in the class hierarchy of the actual class of the object.

We now present the object system interface.[1] Like in Oaklisp, there are two fundamental predefined classes: we call them `Object` and `Class`.[2] Classes and instances are both just *claject*s. For convenience, there are three forms for creating clajects, corresponding to their use as a class, both a class and an instance, or just an instance.

⟨*Claject Construction*⟩≡
```
; for classes of class Class
(new-class %class-id (%super-exp ...)
  ([%slot-id %slot-class-exp
     %slot-deflt-exp ...]
   ; zero or one default exp
   ...)
  (%initializer-exp ...))

; for classes of other classes
(new-claject %class-id %class-exp
   (%super-exp ...)
   ([%slot-id %slot-class-exp
      %slot-deflt-exp ...] ...)
   (%initializer-exp ...)
   %rand ...)

; for instances which are not classes
(new-inst %class-exp %rand ...)
```

Classes require an identifier (used for casting) super-classes, slot declarations and instance initializers. Instances require a class and operands for their initialization. Slot declarations consist of an identifier, a class (used for run-time type-checks) and a default value. Both the class and default value expressions may refer to the fully allocated actual claject as `this`. They may also access slots which are declared earlier in the list.

Primitive classes include `Boolean`, `Integer`, `Procedure`, `Pair`, `List` and `EmptyList`.

⟨*Claject Relationships*⟩≡
```
(eq? %class1 %class2)
(class-of %inst)
(subtype-of? %subtype %super-type)
(isa? %inst %class)
(cast %inst-exp %class-id)
```

The test for class identity is `eq?`. The `cast` operator accesses the subobjects of an object. Casting is permanent, in the sense that the new object has no recollection of its former class. Temporary casts can be defined as a form of method call.

⟨*Slot Access*⟩≡
```
(meta-slot-ref %inst-exp %var %level)
(meta-slot-set! %inst-exp %var %val %level)
(slot-ref %inst-exp %var)
(slot-set! %inst-exp %var %val)
```

The `%level` indicates one less than the number of is-a? links up to where the slot is declared. For slot-ref and slot-set!, it defaults to zero.

⟨*Generic Functions*⟩≡
```
(generic %multimethod ...)
(memo-generic %multimethod ...)
(extend-generic %generic %multimethod ...)
(extend-generic! %generic %multimethod ...)
```

`generic` is the standard form for function creation. `memo-generic` is a memo-ized version. It ensures that for each set of inputs, only one output is generated. It is required because `eq?` is the test of class equality, and we wish to have equivalent applications return equal classes. `extend-generic` is a functional function extension mechanism. `extend-generic!` is the standard imperative add-multimethod operation. Function call is simply application.

⟨*Multimethods and Initializers*⟩≡
```
(multimethod ([%var %type] ...)
  %body0 %body1 ...)
(initializer ([%var %type] ...)
  %body0 %body1 ...)

; within initializers
(initialize %inst %rand ...)
```

[1] The implementation is available on the internet at http://www.cs.indiana.edu/hyplan/sganz/publications/sfp00/code.tar.gz .

[2] Identifiers referring to classes will be captialized.

Both multimethods and initializers are defined as a sequence of parameter declarations followed by a sequence of body expressions. The body of an initializer may refer to the actual claject as `this`.

When an instance is created, its class is searched for the most appropriate initializer (for the operands provided). Within an initializer, `initialize` may be called on the subobjects of an object to initialize them. In either situation, if no initializer is found a default initialization is performed. The default initializer will yield to custom initializers for subobject initialization.

## 3 Building Modular Interpreters

We provide only the beginning functionality of a modular interpreter. Several comments regarding the code are in order. For simplicity, primitives are handled as any other language form. At the cost of some extra complexity, logic for handling various primitive applications can be shared. Error-checking primitive applications based on the number of operands could then also be easily added. The `return` and `extend` operators could be treated as macros which take care to capture `language`, but we choose to leave them in expanded form.

We use a three-level structure, making use of metaclasses. The class `Language` is the broadest language specification, defining the necessary components of any language, including its computational metalanguage. Its subclasses are specifications of more complex languages which may define additional computational operators. Instances of these are languages, whose subobject structure matches the language specification class hierarchy. Treated as classes, languages specify the form of expressions, and are subclassed by the various expression types. Notice that if there were to be a subclass link between languages (none is needed), it would go in the opposite direction of the language specification links, since any expression of a simpler language can be used where an expression of a more complex language is required. The expression types are defined as mixins so that they can be reused as subclasses of various languages. The usage of mixins for writing extensible programs has also been studied by Findler and Flatt [5].

`Language` represents the class of interpreted languages, and so must specify the parsing and evaluation required of any language. Recursions in `parse` will require the use of the `parse` procedure from the language actually being parsed, not the language layer at which any given form and its corresponding parser clause were contributed. Similarly for `eval-exp`, recursions will require the use of the `eval-exp` procedure and computational metalanguage operators from the language actually being evaluated. Thus, the provided definitions of `parse` and `eval-exp`, called `make-parse` and `make-eval-exp`, are relative to the actual language. That language is available as `this` when `Language` is instantiated. Because previously declared slots are available to default-value expressions of later ones, `make-parse` and `make-eval-exp` can be fed the actual language to yield `parse` and `eval-exp`. `Language` also specifies the requirement of `return`, `extend` and `run` procedures for any language. `return` and `extend` are assumed to satisfy the monadic laws [13]. Computations are represented as monadic values. The `return` procedure creates a monadic value representing a trivial computation from a term. The `extend` procedure composes a procedure expecting a value and returning a monadic value with an initial monadic value, yielding a composite monadic value. The `run` procedure undoes the encoding that is required to create a monadic value.

The `go` procedure is used to run the interpreter. For any language, it composes the `run`, `eval-exp` and `parse` procedures.

⟨*General Interpreter Elements*⟩≡

```
(define Language
  (new-class Language (Object)
    ([make-parse Procedure]
     [make-eval-exp Procedure]
     [parse Procedure
       ((slot-ref this make-parse) this)]
     [eval-exp Procedure
       ((slot-ref this make-eval-exp) this)]
     [return Procedure]
     [extend Procedure]
     [run Procedure])
    ()))

(define go
  (generic
    (multimethod ([language Language])
      (compose
        (slot-ref language run)
        (slot-ref language eval-exp)
        (slot-ref language parse)))))
```

### 3.1 Basic Language

Computations in the most basic interpreter are modeled on the identity monad. That monad is specified such that both `return` and `extend` are the identitity function. Since no encoding is imposed, the `run` procedure is also the identity function.

⟨*Identity Monad*⟩≡

```
(define return-id (lambda (t) t))
(define extend-id (lambda (f) f))
(define run-id (lambda (m) m))
```

The class `Basic-language` represents basic interpreted languages. This is equivalent to `Language`, the class of interpreted languages, because the `*` form which is added at this level requires no additonal operators in the computational metalanguage. In practice, however, all instances of `Basic-language` will have `parse` and `eval-exp` functions able to handle the `*` form. The class includes basic interpreted languages such as `Basic-expression` defined below, and basic language subobjects of more specialized languages such as those belonging to `Signal-language`.

Our basic language contains two forms: integer literals and binary multiplications, represented as two subclasses of a basic language claject. Thinking in terms of extensibility, such a basic language claject need not be an instance of `Basic-language` directly, but may be an instance of any of its subclasses. Thus, we define integer literals and multiplication expressions as mixins, *i.e.*, functions from super classes to derived classes [1]. In the case of multiplication expressions, which have recursive subparts, the language parameter serves another purpose. It is used to declare the type of the operand subexpressions.

⟨*Basic Language*⟩≡

```
(define Basic-language
  (new-class Basic-language (Language)
    () ()))

(define mixin-Int-lit
  (memo-generic
    (multimethod ([language Basic-language])
      (new-class Int-lit (language)
        ([int Integer]) ())))))

(define mixin-Mult-exp
  (memo-generic
    (multimethod ([language Basic-language])
      (new-class Mult-exp (language)
        ([rand1 language]
         [rand2 language])
        ())))))

(define Basic-expression
  (new-claject Basic-expression Basic-language
    (Object) () ()
    ⟨Basic make-parse⟩
    ⟨Basic make-eval-exp⟩
    return-id extend-id run-id))
```

When we instantiate `Basic-language` as `Basic-expression`, we create a particular basic language with procedures for parsing and evaluating nested multiplication expressions, and a core computational metalanguage. That basic language is itself a class which is given `Object` as a superclass. It specifies no slots itself, as these will be provided by its subclasses through the mixins presented above. The identity monad is all that is required for the core computational metalanguage.

As we have explained, `make-parse` and `make-eval-exp` are relative to an actual language. In this case, the actual language is some `Basic-language`. For `make-parse`, the resulting procedure is responsible for creating expressions of that language, using the mixins. Recursions are to the `parse` procedure of that language.

⟨*Basic* `make-parse`⟩≡

```
(generic
  (multimethod ([language Basic-language])
    (lambda (x)
      (cond
        [(integer? x)
         (new-inst (mixin-Int-lit language)
           x)]
        [(and (pair? x) (eq? (car x) '*))
         (new-inst (mixin-Mult-exp language)
           ((slot-ref language parse)
            (cadr x))
           ((slot-ref language parse)
            (caddr x)))]
        [else
          (error 'parse
            "Couldn't parse ~s" x)]))))
```

For `make-eval-exp`, the resulting procedure uses the mixins to declare its parameters, and looks to the actual language for definitions of `return` and `extend`, as well as for the recursions.

⟨*Basic* `make-eval-exp`⟩≡

```
(memo-generic
  (multimethod ([language Basic-language])
    (generic
      (multimethod
        ([int-lit (mixin-Int-lit language)])
        ((slot-ref language return)
         (slot-ref int-lit int)))
      (multimethod
        ([mult-exp (mixin-Mult-exp language)])
        (((slot-ref language extend)
          (lambda (arg1)
            (((slot-ref language extend)
              (lambda (arg2)
                ((slot-ref language return)
                 (* arg1 arg2))))
             ((slot-ref language eval-exp)
              (slot-ref mult-exp rand2)))))
         ((slot-ref language eval-exp)
          (slot-ref mult-exp rand1)))))))
```

### 3.2 Signal Language

The type `ErrorComp` is introduced to represent computations under the error monad. It has two variants, `Ok`, which holds a value, and `Bad`, which holds an error code. We obtain the error monad by applying the error monad transformer, which takes the form of a sequence of transformer functions, to elements of the identity monad. The error monad transformer is as presented by Liang, Hudak and Jones [9].

A specification of a new operator for the computational metalanguage is now provided. The `signal-errt` precedure does not require any arguments, but other operators may require any elements of the previous computational metalanguage. The resulting operator takes an error code and builds a `Bad` record.

⟨*Error Monad Transformer*⟩≡

```
(define ErrorComp
  (new-class ErrorComp (Object)
    () ()))

(define Ok
  (new-class Ok (ErrorComp)
    ([val Object]) ()))

(define Bad
  (new-class Bad (ErrorComp)
    ([code Integer]) ()))


(define return-errt
  (lambda (return-prev)
    (compose return-prev
      (lambda (x) (new-inst Ok x)))))

(define extend-errt
  (lambda (return-prev extend-prev)
    (lambda (f)
      (lambda (in-m)
        ((extend-prev
          (generic
```

```
          (multimethod ([ok Ok])
            (f (slot-ref ok val)))
          (multimethod ([bad Bad])
            (return-prev bad))))
       in-m)))))

(define run-errt
  (lambda (run-prev)
    (generic
      (multimethod ([ok Ok])
        (run-prev
         (slot-ref ok val)))
      (multimethod ([bad Bad])
        (error 'eval "code: ~s~n"
          (slot-ref bad code))))))

(define signal-errt
  (lambda () (lambda (x) (new-inst Bad x))))
```

Our language will be extended with two new forms: division expressions and error expressions. To support these, all succeeding languages must provide a `signal-err` operator in additon to the standard monadic operators. The mixin for division expressions is similar to that for multiplication expressions. The mixin for error expressions is similar to that for integer literals.

Next, we wish to create a `Signal-expression` class as we did for `Basic-expression`. We will require a `Basic-expression` class in order to accomplish this, since such a claject will have `parse` and `eval-exp` clauses for the rest of our language, as well as computational metalanguage operators that we will need in order to obtain our own metalanguage using the monad transformer. Thus, we first define `Signal-expression` relative to a `Basic-language` which we call `prev-language`, and then provide it with the only basic language we have available, `Basic-expression`. In some cases, *e.g.*, call-by-name vs. call-by-value semantics for the $\lambda$-calculus, there might be multiple reasonable choices for `prev-language`.

We use the error monad transformer to modify the monadic operators from the identity monad, and provide our new operator, `signal-err`. Had there been any other operators in the previous language, we would lift them to the new language [9].

⟨*Signal language*⟩≡

```
(define Signal-language
  (new-class Signal-language
    (Basic-language)
    ([signal-err Procedure])
    ()))

(define mixin-Div-exp
  (memo-generic
    (multimethod ([language Signal-language])
    (new-class Div-exp (language)
      ([rand1 language]
       [rand2 language])
      ()))))

(define mixin-Error-exp
  (memo-generic
    (multimethod ([language Signal-language])
    (new-class <error> (language)
      ([code Integer])
      ()))))
```

```
(define new-Signal-language
  (memo-generic
    (multimethod
      ([prev-language Basic-language])
      (new-claject Signal-expression
        Signal-language (Object) () ()
        ⟨Signal make-parse⟩
        ⟨Signal make-eval-exp⟩
        (return-errt
          (slot-ref prev-language return))
        (extend-errt
          (slot-ref prev-language return)
          (slot-ref prev-language extend))
        (run-errt
          (slot-ref prev-language run))
        (signal-errt)))))

(define Signal-expression
  (new-Signal-language Basic-expression))
```

The parse procedure makes use of `prev-language` when it discovers that none of its own clauses apply to the input expression. It uses the clauses from the parser of the old language, but views them as directing the construction of language forms in the current language. Thus, we use `make-parse` and apply it to `language`.

⟨*Signal make-parse*⟩≡

```
(generic
  (multimethod ([language Signal-language])
    (lambda (x)
      (cond
        [(and (pair? x) (eq? (car x) '/))
         (new-inst (mixin-Div-exp language)
           ((slot-ref language parse)
            (cadr x))
           ((slot-ref language parse)
            (caddr x)))]
        [(and (pair? x) (eq? (car x) 'error))
         (new-inst (mixin-Error-exp language)
           ((slot-ref language parse)
            (cadr x)))]
        [else
          (((slot-ref prev-language
              make-parse)
            language)
           x)]))))
```

The interpreter procedure obtains the previous interpreter from `prev-language`, and extends it with additional multimethods for the language extension. Just as with the parser above, we must rebuild the interpreter using `language`. Thus, we make use of the new computational metalanguage, and the new interpreter in recursions. By functionally extending the old interpreter, we leave the original intact so that we can use it alongside the new one without additional overhead.

⟨*Signal make-eval-exp*⟩≡

```
(generic
  (multimethod ([language Signal-language])
    (extend-generic
      ((slot-ref prev-language make-eval-exp)
       language)
      (multimethod
        ([div-exp (mixin-Div-exp language)])
        (((slot-ref language extend)
          (lambda (arg1)
            (((slot-ref language extend)
              (lambda (arg2)
```

```
              (if (zero? arg2)
                ((slot-ref language
                   signal-err)
                 9002)
                ((slot-ref language return)
                 (/ arg1 arg2)))))
           ((slot-ref language eval-exp)
            (slot-ref div-exp rand2)))))
       ((slot-ref language eval-exp)
        (slot-ref div-exp rand1))))
    (multimethod
      ([error-exp
         (mixin-Error-exp language)])
      (((slot-ref language extend)
        (lambda (code-arg)
          ((slot-ref language signal-err)
           code-arg)))
       ((slot-ref language eval-exp)
        (slot-ref error-exp code)))))))
```

In summary, each interpreter module contains the following components:

- Reference to previous interpreter module.
- Reference to monad transformer being added to semantics.
- Language specification (derived from previous language specification), and including any new semantic operators as slots.
- Mixins defining syntax of new language forms.
- Expression specification (in terms of previous expression specification), and including extension of previous generic functions for parsing and evaluation, as well as transformed/lifted monadic operators from the previous language, and any new operators.

### 3.3 Full Interpreter

Although we have presented only the first two interpreter modules, we have found that this approach scales up well. The system has been extended to provide extensive functionality described below.[3]

The following monad transformers were used:
    with custom operators

error monad
    signal-err
environment monad
    current-env
    modify-env
store monad
    current-sto
    modify-sto!
continuation monad
    callcc

The following interpeter modules were used:
    with monad transformer applied
    and language features added

basic
    applies identity monad as base
    lit form
    if form
    primitive application form
    * primitive

signal
    applies error monad transformer
    error primitive
    / primitive
    errors are signalled on div by zero, etc.

read-only stack
    applies environment monad transformer
    (environment holds values)
    let form
    variable reference form

read-write stack
    (environment now holds locations)
    applies store monad transformer
    set! form

read-write stack, heap
    reapplies store monad transformer
    box primitive
    unbox primitive
    set-box! primitive

read-write stack, heap, continuations
    applies continuation monad transformer
    letcc form
    throw form

## 4 Conclusion

We have explored the use of an object-oriented extension of Scheme for writing modular interpreters for mostly functional languages. We first presented our object system, which features generic functions with multimethods, multiple inheritance and metaclasses. We then constructed the first several modules of an interpreter, using techniques that could easily be extended for any number of modules.

Our effort has largely been successful. However, there is room for improvement. Notice that although we used mixins to represent the syntax of forms in a language, the interpreter modules themselves are each hard-wired to the previous module. We were able to parameterize over the actual pre-

[3] Again, see http://www.cs.indiana.edu/hyplan/sganz/publications/sfp00/code.tar.gz for the full implementation.

vious language, but the previous language specification must be fixed. This is primarily because such modularity would require automated techniques for lifting monadic operators. Even then, the initialization of abstract superclasses would be complex. Monadic operators (other than `return` and `extend`) would be defaulted to their non-lifted values through the language specification. An additional argument (the list of monad transformers to be applied) would have to be added to language constructors. It would be forwarded up the hierarchy and would, upon reaching a superclass with such a monadic operator, be used to generate lift procedures and apply them to the operator last-in, first-out, updating it in the language instance.

Our interpreter thus falls short of the claims of complete independence and interchangeability made by Cartwright and Felleisen [2], although there are some interesting similarities between our methods. There are several reasons for the discrepancy. First, they use a largely untyped language, while ours is typed, albeit dynamically. The typing of expression components is an element of the interpreter that we need to update during any extension. Second, they forgo the monadic approach to semantics (although their programs are written in a monadic style) in favor of the more straightforward one of sending a message, including an evaluation context, along with a list of resources to an administrative routine. They thus seem to avoid the complex issue of lifting operators which was a focus of Liang, Hudak and Jones. A possible indication of a shortcoming of Cartwright and Felleisen's approach is that they do not treat environments as a modular extension, but as an inherent part of any interpreter. A third reason is their use of mutation of global variables, which has the effect that the extensions to their interpreter cannot coexist with the original.

Another disappointing aspect of this interpeter is the isolation of the mixins from the rest of the language. One would prefer to see those mixins combined and stored as part of the language, as the object portion of a syntax functor. This would in turn pave the way for the introduction of catamorphisms for the definition of the interpreter and anamorphisms for the definition of the parser, making the recursions implicit. It will be the subject of future research.

## References


[1] Gilad Bracha and William Cook. Mixin-based inheritance. In *ECOOP/OOPSLA '90*, pages 303–311, 1990.

[2] Robert Cartwright and Matthias Felleisen. Extensible denotational language specifications. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 244–272. Springer-Verlag, April 1994.

[3] William R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, New York, NY, 1991.

[4] Luc Duponcheel. Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters. Research Report (draft), November 1995.

[5] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 94–104. ACM, June 1999.

[6] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 52–61, New York, N.Y., June 1993. ACM Press.

[7] Shriram Krishnamurthi and Matthias Felleisen. Toward a formal theory of extensible software. In *Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering (FSE-98)*, volume 23, 6 of *Software Engineering Notes*, pages 88–98, New York, November 3–5 1998. ACM Press.

[8] Kevin J. Lang and Barak A. Pearlmutter. Oaklisp: an object-oriented dialect of scheme. *Lisp and Symbolic Computation: An International Journal*, 1(1):39–51, May 1988.

[9] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California*, pages 333–343. ACM, January 1995.

[10] Jonathan G. Rossie Jr., Daniel P. Friedman, and Mitchell Wand. Modeling subobject-based inheritance. In Pierre Cointe, editor, *ECOOP '96 — object-oriented programming: 10th European Conference, Linz, Austria, July 8–12, 1996: proceedings*, volume 1098 of *Lecture Notes in Computer Science*, pages 248–274. Springer, 1996.

[11] A. Snyder. Common objects: an overview, 1986.

[12] Guy L. Steele, Jr. Building interpreters by composing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT*

*Symposium on Principles of Programming Languages, Portland, Oregon*, pages 472–492. ACM, January 1994.

[13] P. Wadler. Monads for functional programming. *Lecture Notes in Computer Science*, 925:24–??, 1995.