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 Dupontbeau [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 pre-existing 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 interpreter. 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]. Dupontbeau 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 Felleisen [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 metalinguage, 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 type-class in place of the type-class mechanism itself.

*This work was supported in part by the National Science Foundation under grant CDA-8912614. Author's address: Department of Computer Science, Indiana University, Bloomington, Indiana 47405. sgmeacs.indiana.edu.
†This work was supported in part by the National Science Foundation under grant CCR-9633100. Author's address: Department of Computer Science, Indiana University, Bloomington, Indiana 47405. dfriedracs.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 a 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 Oakdip [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. Oakdip 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 meta-classes, 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 Oakdip, there are two fundamental pre-defined classes: we call them Object and Class.2 Classes and instances are both just clajets. For convenience, there are three forms for creating clajets, corresponding to their use as a class, both a class and an instance, or just an instance.

(\texttt{Class \textup{Construction}}) \equiv
\begin{align*}
&\text{for classes of class Class} \\
&\text{(new-class \{\texttt{\%class-id \%super-exp \ldots}\})} \\
&\quad\text{\{\texttt{\%slot-id \%slot-class-exp}} \\
&\quad\quad\text{\{\texttt{\%slot-default-exp \ldots}\}} \\
&\quad\quad; \text{\texttt{\%initialize-exp \ldots}}) \\
&\text{; for classes of other classes} \\
&\text{(new-clajet \{\texttt{\%class-id \%class-exp}\})} \\
&\quad; \text{\texttt{\%initialize-exp \ldots}}
\end{align*}

1The implementation is available on the internet at \url{http://www.cs.indiana.edu/~hyplan/sganx/publications/ sf900/code.tar.gz}.

2Identifiers referring to classes will be capitalized.

(\texttt{\%super-exp \ldots})
(\texttt{[\%slot-id \%slot-class-exp}}
\quad\text{\{\texttt{\%slot-default-exp \ldots}\}} \\
\quad\text{(\%initialize-exp \ldots})}
\text{\%rand \ldots})
\begin{align*}
&\text{for instances which are not classes} \\
&\text{(new-inst \{\texttt{\%class-exp \%rand \ldots}\})}
\end{align*}

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 clajet as this. They may also access slots which are declared earlier in the list.

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

(\texttt{\%Class \textup{Relationships}}) \equiv
\begin{align*}
&\text{\texttt{(eq? \%class1 \%class2)}} \\
&\text{\texttt{(class-of \%inst)}} \\
&\text{\texttt{(\%sub-type \%super-type)}} \\
&\text{\texttt{(\%is? \%inst \%class)}} \\
&\text{\texttt{(cast \%inst \%exp \%class-id)}}
\end{align*}

The test for class identity is \texttt{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.

(\texttt{\%Slot Access}) \equiv
\begin{align*}
&\text{\texttt{(meta-slot-ref \%inst \%exp \%var \%level)}} \\
&\text{\texttt{(meta-slot-set! \%inst \%exp \%var \%val \%level)}} \\
&\text{\texttt{(slot-ref \%inst \%exp \%var)}} \\
&\text{\texttt{(slot-set! \%inst \%exp \%var \%val)}}
\end{align*}

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.

(\texttt{\%Generic \textup{Functions}}) \equiv
\begin{align*}
&\text{\texttt{(\%generic \%multimethod \ldots)}} \\
&\text{\texttt{(memo-generic \%multimethod \ldots)}} \\
&\text{\texttt{(\%extend-generic \%\texttt{\%\%\%generic \%multimethod \ldots}}}} \\
&\text{\texttt{(\%extend-generic! \%\texttt{\%\%\%generic \%multimethod \ldots}}}}
\end{align*}

generic is the standard form for function creation. memo-generic is a memoized 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.

(\texttt{\%Multimethods \textup{and \%Initializers}}) \equiv
\begin{align*}
&\text{\texttt{(multimethod ([\%var \%type \ldots])}} \\
&\quad\text{\texttt{\%body0 \%body1 \ldots}}} \\
&\quad\text{\texttt{(initialize ([\%var \%type \ldots})}} \\
&\quad\quad\text{\texttt{\%body0 \%body1 \ldots)}} \\
&\quad; \text{\texttt{within initializers}} \\
&\quad\text{\texttt{(initialize \%inst \%rand \ldots)}}
\end{align*}
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 object 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 compose 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.

\[(\text{General Interpreter Elements})\]

\[
\begin{aligned}
&\text{define Language} \\
&\quad\{\text{new-class Language (Object)} \\
&\quad\quad\{\text{make-parse Procedure} \\
&\quad\quad\quad\text{[parse Procedure}} \\
&\quad\quad\quad\quad\{\text{slot-ref this make-parse} this\}\} \\
&\quad\quad\quad\text{[eval-exp Procedure} \\
&\quad\quad\quad\quad\quad\{\text{slot-ref this make-eval-exp} this\}\} \\
&\quad\quad\quad\text{[return Procedure} \\
&\quad\quad\quad\text{[extend Procedure} \\
&\quad\quad\quad\text{[run Procedure} \\
&\quad\quad\quad\} \\
&\quad\} \\
&\text{(}))}
\end{aligned}
\]

\[
\text{define go}
\]

\[
\begin{aligned}
&\quad\{\text{generic} \\
&\quad\quad\{\text{multimethod (language language)} \\
&\quad\quad\quad\{\text{compose} \\
&\quad\quad\quad\quad\{\text{slot-ref language run} \\
&\quad\quad\quad\quad\quad\{\text{slot-ref language eval-exp} \\
&\quad\quad\quad\quad\quad\quad\{\text{slot-ref language parse})\})))
\quad\quad\} \\
&\quad\quad\} \\
&\quad\}
\end{aligned}
\]

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 identity function. Since no encoding is imposed, the run procedure is also the identity function.

\[(\text{Identity Monad})\]

\[
\begin{aligned}
&\text{define return-id (lambda (t) t)} \\
&\text{(define extend-id (lambda (t) t))} \\
&\text{(define run-id (lambda (m n))}
\end{aligned}
\]

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 additional 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 object. Thinking in terms of extensibility, such a basic language...
clobject 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.

\[\text{Basic Language} \equiv \]
\[\text{(define Basic-language}
\text{(new-class Basic-language Language)}
\text{(lambda (x) ()))}
\text{)}\]
\[\text{(define mixin-Int-lit}
\text{(semo-generic}
\text{(multimethod \{[language Basic-language]\})}
\text{(new-class Int-lit (language)}
\text{([Int Integer]) (()))))}
\text{)}\]

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.

\[\text{Basic make-parse} \equiv \]
\[\text{(generic}
\text{(multimethod \{[language Basic-language]\})}
\text{(lambda (x) (cond}
\text{[(integer? x) (new-inst (mixin-Int-lit language) x)]}
\text{[(and (pair? x) (eq? (car x) '+)) (new-inst (mixin-Mult-exp language) ((slot-ref language parse) (cadr x))
\text{((slot-ref language parse) (caddr x)))]}
\text{[else (error \"parse \"x))))))}
\text{)\text{)}}}

\[\text{For make-eval-exp, the resulting procedure uses the mixins to declare its parameters, and looks to the actual language for definitions of return and extend, as well as for the recursions.}\]
\[\text{[Basic make-eval-exp]}\equiv \]
\[\text{(semo-generic}
\text{(multimethod \{[language Basic-language]\})}
\text{(generic}
\text{(multimethod \{[Int-lit (mixin-Int-lit language)]\})
\text{(slot-ref language return)
\text{((slot-ref int-lit int)))}
\text{(multimethod \{[Mult-exp (mixin-Mult-exp language)]\})
\text{(((slot-ref language extend)
\text{lambda (arg1)
\text{(((slot-ref language extend)
\text{lambda (arg2)
\text{((slot-ref language eval-exp)
\text{((slot-ref mult-exp rand2))))))))
\text{((slot-ref language eval-exp)
\text{((slot-ref mult-exp rand1)))))))))}}
\text{)}}
\text{)}}

3.2 Signal Language

The type ErrorComp is introduced to represent computations under the error monad. It has two variants, 0k, 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 procedure 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.

\[\text{[Error Monad Transformer]}\equiv \]
\[\text{(define ErrorComp}
\text{(new-class ErrorComp (Object)}
\text{(lambda (x) ()))}
\text{))\]
\[\text{(define Ok}
\text{(new-class Ok (ErrorComp)}
\text{((lambda (x) ()))
\text{))}\]
\[\text{(define Bad}
\text{(new-class Bad (ErrorComp)}
\text{((lambda (x) ()))
\text{))}\]
\[\text{(define return-errt}
\text{((lambda (return-prev)
\text{compose return-prev)
\text{lambda (x) (new-inst 0k x)))))}
\text{))}\]
\[\text{(define extend-errt}
\text{((lambda (return-prev extend-prev)
\text{lambda (df)
\text{lambda (in-r)
\text{((extend-prev)
\text{generate
\text{))))))}}
\text{))}\]
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 addition 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 in order to accomplish this, since such a class 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].

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.
(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
     [mix-in 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:

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

continuation monad
callcc

The following interpreter modules were used:

with monad transformer applied
and language features added

basic

applies identity monad as base

nit form

if form

primitive application form

* primitive

signal

applies error monad transformer

error primitive

/ primitive

ersors 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-
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 forget 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 interpreter 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


