Type Classes Without Types *

Ronald Garcia Andrew Lumsdaine

Open Systems Lab
Indiana University
Bloomington, IN 47405
{garcia,lums}@cs.indiana.edu

Abstract

Data-directed programs consist of collections of generic functions, functions whose underlying implementation differs depending on properties of their arguments. Scheme's flexibility lends itself to developing generic functions, but the language has some shortcomings in this regard. In particular, it lacks both facilities for conveniently extending generic functions while preserving the flexibility of ad-hoc overloading techniques and constructs for grouping related generic functions into coherent interfaces. This paper describes and discusses a mechanism, inspired by Haskell type classes, for implementing generic functions in Scheme that directly addresses the aforementioned concerns. Certain properties of Scheme, namely dynamic typing and an emphasis on block structure, have guided the design toward an end that balances structure and flexibility. We describe the system, demonstrate its function, and argue that it implements an interesting approach to polymorphism and, more specifically, overloading.

1. Introduction

Data-directed programs consist of collections of *generic functions*, functions whose underlying implementation differs depending on properties of their arguments. In other words, a generic function is *overloaded* for different argument types. Data-directed style appears often in Scheme programs, even in the Scheme standard library. The standard generic arithmetic operators include functions such as `+` and `*`, which exhibit different behavior depending on what kind of arguments they are applied to. For example, applying `+` to two integers yields an integer value; adding two complex values, on the other hand, yields a complex value. A binary version of `+` could be implemented with the following general form:

```
(define +
  (lambda (a b)
    (cond
    [(and (integer? a) (integer? b))
     (integer-+ a b)]
     [(and (complex? a) (complex? b))
      (complex-+ a b)]
     ...
     [else (error "invalid arguments")])))
```

The body of `+` is simply a `cond` expression that tests its operands for various properties and dispatches to the implementation upon finding a match. Assuming specific implementations of addition for integers and complex numbers, the function dispatches to integer addition when the operands are integers, and complex numbers when the operands are complex. [1]

For all their benefits, generic functions implemented using `cond` as above have their shortcomings. Such functions are not meant to be extended to support new types of arguments. Nonetheless, such a function may be extended at the top-level using ad-hoc means as in the following:

```
(define +
  (let ([old-+ +])
    (lambda (a b)
      (cond
      [(and (my-number? a) (my-number? b))
       (my-+ a b)]
       [else (old-+ a b)]))))
```

A function may also be extended in a manner that limits the extension to the current lexical scope, as in the following:

```
(let ([+
       (let ([old-+ +])
         (lambda (a b)
           (cond
           [(and (my-number? a) (my-number? b))
            (my-+ a b)]
            [else (old-+ a b)])))])
  (+ my-number-1 my-number-2))
```

The above examples assume a user-defined number, of which `my-number-1` and `my-number-2` are instances, and a `my-number?` predicate that tests for such numbers. Both versions of `+` can handle these new numbers. Although the second example only introduces the new `+` in the scope of the `let` expression, the function could be returned as a value from the expression and subsequently used in other contexts.

These methods of extending `+` are ad-hoc. They don't directly capture the intent of the programmer, and much of the content is boiler-plate code. Another issue with this style of extending data-directed functions is that it does not respect the grouping of related functions. For example, the `+` operator is just one of a group of arithmetic operators that includes `*`, `-`, and `/` as well, and in general they should be introduced and extended together. Using the above method of introducing overloads, one must manually duplicate the

* This material is based on work supported by NSF grant EIA-0131354 and by a grant from the Lilly Endowment.

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

[1] This model disregards the possible coercion of arguments to match each other because such a mechanism is outside the scope of this work.

idiom for each operator, resulting in duplicate boilerplate code and no intentional structuring of the set of operators.

The Haskell [Pey03] language community has previously investigated overloading in the context of a statically typed language and as their answer to the problem produced the *type class* facility [WB89], which we describe later. Type classes are an elegant, effective approach to overloading and have spawned significant research that has advanced their capabilities [NT02, Jon00, CHO92].

This paper describes a language extension for Scheme that supports the implementation of groups of generic functions and their overloads. This system is heavily inspired by Haskell's type classes, but is designed to function in a latently typed language, where types appear as predicates on values. For that reason, we consider ours to be a *predicate class* system.

In order to fit with Scheme, this system differs from Haskell's type classes in some significant ways. Haskell is solely interested in dispatch based on static type information. In contrast, the ad-hoc method of constructing and extending generic functions can dispatch on arbitrary predicates, including standard predicates such as `number?` and `char?`, as well as user-defined predicates such as `my-number?` from the earlier examples. The described system also supports overloading based on arbitrary predicates. Also, whereas Haskell emphasizes compile-time type checking, error-checking is subservient to flexibility in this model. The overloading mechanism described here eschews the conservative practice of signaling errors before they are encountered at run time.

The combination of block structure, lexical scoping, and referential transparency plays a significant role in Scheme programs. Some of the previously discussed ad-hoc methods show how overloading can be performed in Scheme and how those methods fall short in lexical contexts. The predicate class system we present directly supports overloading functions under such circumstances.

Our overloading mechanism was implemented for Chez Scheme using the syntax-case macro system [DHB92, Dyb92], an advanced macro expansion system provided by some popular Scheme implementations that combines hygienic macro expansion [KFFD86] with controlled identifier capture. Because our system is implemented as macros, its semantics can be described in terms of how the introduced language forms are expanded (See Section 6).

2. Contributions

The contributions of this paper are as follows:

- A language mechanism for overloading is described, inspired by the type class model but modified to better match the capabilities and the philosophy of Scheme. Scoped classes and instances that allow both classes and instances to be shadowed lexically is an interesting point in the design space. Furthermore, expressing this facility in the context of a dynamically typed language allows some interesting design options and tradeoffs that are not available to statically typed languages.
- The described dynamic dispatch model combines the flexibility of ad-hoc techniques available in Scheme with a more structured mechanism for overloading functions. Previous mechanisms for overloading in Lisp and Scheme have pointed toward a relationship to objects and object-oriented programming. Our system supports dispatch based on arbitrary runtime properties. Furthermore, the predicate class model groups related generic functions into extensible interfaces.
- A point of comparison is provided between the overloading mechanisms expressed in statically typed Haskell and dynamically typed Common Lisp traditions.

3. A Brief overview of Haskell Type Classes

Haskell [Pey03] is a statically typed functional programming language, featuring Hindley/Milner-style type inference [Mil78, DM82] and its associated flavor of parametric polymorphism. Haskell also, however, supports a form of *ad-hoc* polymorphism, or overloading, in the form of type classes [WB89], which contribute substantial expressive power to the language. In order to introduce the concepts involved, and to provide a point of comparison, we briefly describe the type class system.

```
class  Eq a  where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
   x /= y         =  not (x == y)
   x == y         =  not (x /= y)

instance Eq Integer where
  x == y         =  x `integerEq` y

instance Eq Float where
  x == y         =  x `floatEq` y

elem :: (Eq a) => a -> [a] -> Bool
x `elem`  []     = False
x `elem` (y:ys)  = x==y || (x `elem` ys)
```

Figure 1. Haskell type classes in action

Consider the problem of specifying and using operators for numeric types, specifically the equality operator. Figure 1 illustrates how the equality operator is specified for Haskell in its Standard Prelude. First a *type class* is introduced. A type class is an interface that specifies a set of *class operators*, generic functions associated with a particular type class. The above type class definition essentially says "for all types `a` that belong to the class `Eq`, the operators `==` and `/=` are overloaded with values of the specified type signatures." The `Eq` class defines default implementations for `==` and `/=`, however in order to use them, a type must be explicitly declared to overload the type class functions. This role is played by *instance declarations*. An instance declaration declares membership in the type class and implements *instance methods*, specialized overloads of the class operators. For example, the first instance declaration for `Integer`s declares that `Integer` is a member of the `Eq` class, and provides an explicit overload for `==`. The `==` operator for `Integer` values is implemented in terms of a hypothetical `integerEq` operator defined solely for integers. [2] An analogous instance for floats is also presented. Both instance declarations inherit the default `/=` method, which will call the specific `==` overload associated with the type. In fact one may legally omit the `==` implementation as well, but then a call to either operator yields an infinite recursion. Finally, the `elem` function, analogous to Scheme's `member`, is presented. This generic function is not part of the `Eq` type class, yet still relies upon it. Its type, `(Eq a) => a -> [a] -> Bool` is *qualified* with `Eq` and essentially says "`elem` is overloaded for all types `a` that belong to `Eq`, in which case its type is `a -> [a] -> Bool`."

4. Language Description

The predicate class mechanism introduced in this paper forms an embedded language for overloading in Scheme and thus purely extends the existing language. This section introduces and describes the forms with which we extend Scheme to provide type class-like functionality. The extended language syntax is summarized in Figure 2.

[2] In Haskell, a binary function can be called in infix position by enclosing it in single back quotes

```
⟨definition⟩ +→
     (define-class (⟨identifier⟩ ⟨variable⟩+)
       ⟨op-spec⟩+)
   | (define-instance (⟨identifier⟩ ⟨expression⟩+)
       (⟨method-name⟩ ⟨expression⟩)+)
   | (define-qualified ⟨identifier⟩ (⟨identifier⟩+)
       ⟨expression⟩)
   | (define-open-qualified ⟨identifier⟩ (⟨identifier⟩+)
       ⟨expression⟩)

⟨expression⟩ +→
     (let-class ([(⟨identifier⟩ ⟨variable⟩+)
                  ⟨op-spec⟩+])
       ⟨body⟩)

   | (let-instance ([(⟨identifier⟩ ⟨expression⟩+)
                     (⟨method-name⟩ ⟨expression⟩)+])
       ⟨body⟩)

⟨op-spec⟩ → (⟨operator-name⟩ ⟨variable⟩*)
   | [(⟨operator-name⟩ ⟨variable⟩*) ⟨expression⟩]
```

Figure 2. Syntax extensions for type classes in Scheme

4.1 Predicate Classes

A predicate class is a form that establishes an interface for overloading. Predicate classes are introduced using either the `define-class` form, for top-level definitions, or the `let-class` expression, for lexically scoped definitions. The syntax that we use for these constructs is as follows: [3]

```
(define-class (class-name pv ...)
   op-spec
   ...)

(let-class ([(class-name pv ...)
             op-spec ...])
  expr ...)
```

The `define-class` form introduces a new predicate class at the top-level with the name *class-name*. The `let-class` form correspondingly introduces a type class that is visible within the scope of its enclosed body (*expr* ...). The name of the type class is followed by a list of *predicate variables* (*pv* ...). A class's predicate variables determine the number of predicate functions that will be used to establish an instance of a predicate class. The order of the predicate variables matters, and corresponds directly to the order of the predicates that are used to define an instance (as shown in the next section). Whereas Haskell type class instances are determined by the type used in an instance definition, predicate classes are determined by a list of Scheme predicate functions. This corresponds directly to the Haskell extension that supports multiple parameter type classes [PJM97]. Following the name of the class and its list of predicate variables is a list of *class operation specifications*, signified above by *op-spec*. Each operation specification takes one of the following two forms:

```
(op-name sym ...)

[(op-name sym ...) expr]
```

their purpose is to establish the names of the operators belonging to the class, as well as to specify which arguments will be used to determine dispatch based on which predicates. The second syntax for operation specifications illustrates how to supply a *default instance method* for a class operation. Each symbol *sym* marks an argument position for the operation. Any position marked with a predicate variable will be used to determine dispatch to the proper instance method. If a predicate variable is placed in an argument position, then a call to that class operation will use that argument position to test for instance membership: The instance predicate associated with the given predicate variable will be applied to the passed argument. Instances of the class are tested until an instance is found whose predicates return `#t` for each argument position marked with a predicate variable. The dispatch algorithm implies that the order in which instances are declared can affect the instance that a class operator dispatches to. In this regard, the mechanics of dispatch are analogous to the `cond` form of dispatch described earlier.

For example, consider the following rendition of the `Eq` type class in Scheme:

```
(define-class (Eq a)
  [(== a a) (lambda (l r) (not (/= l r)))]
  [(/= a a) (lambda (l r) (not (== l r)))])
```

This definition looks similar to the Haskell equivalent in Figure 1, but there are a few differences. A Haskell type class specification is used for type checking as well as dispatch. The class's type variable would be instantiated and used to ensure that code that calls the class operators is type safe. In the case of the above Scheme code, however, the predicate variable `a` simply specifies how to dispatch to the proper instance of a method. As written, calls to the `==` method determine dispatch by applying the predicate to both arguments. In some cases, however, the underlying implementations all require both arguments to have the same type. Under that assumption, one can optimize dispatch by checking only the first argument: the dispatched-to function is then expected to report an error if the two values do not agree. The following example shows how to implement such a single-argument dispatch:

```
(define-class (Eq a)
  [(== a _) (lambda (l r) (not (/= l r)))]
  [(/= a _) (lambda (l r) (not (== l r)))])
```

In the above code, the second reference to `a` in each of these operations is replaced with the underscore symbol (`_`). Since the underscore is not one of the specified predicate variables, it is ignored. Symbols that do not represent predicates are most useful, however, when dispatch is dependent on argument positions other than the first. For example in the form:

```
(define-class (Eq a)
  [(== _ a) (lambda (l r) (not (/= l r)))]
  [(/= _ a) (lambda (l r) (not (== l r)))])
```

dispatch is determined by the second argument to the operations.

Under some conditions, it is useful to develop a class that dispatches on multiple predicates, rather than two. For example, consider a type class that specifies overloaded operators that operate on vector spaces. A vector space must take into consideration both the sort of vector and scalar types used, and this can be done as follows:

```
(define-class (Vector-Space v s)
  [vector-add v v]
  [scalar-mult s v])
```

Notice that in particular, scalar multiplication takes a scalar as its first argument and a vector as its second. Classes that represent multi-sorted algebras are bound to have one predicate for each sort.

4.2 Class Instances

A class instance is an implementation of overloads for a specified predicate class that is associated with a particular list of Scheme

[3] Throughout the text, code uses square brackets (`[]`) and parentheses (`()`) interchangeably for readability. Several Scheme implementations, including Chez Scheme, support this syntax.

predicates. they are introduced using the `define-instance` form or the `let-instance` expression. The syntax for these constructs is as follows:

```
(define-instance (class-name pred ...)
  (method-name expr) ...)

(let-instance ([(class-name pred ...)
                (method-name expr) ...])
  expr ...)
```

The `define-instance` form introduces a new top-level instance of a previously declared class. The `let-instance` form correspondingly introduces a new instance of a class for the scope of its body (*expr* ...). An instance definition names the referent class followed by a list of Scheme predicates–functions of one parameter that verify properties of objects. Built-in examples include `integer?` and `boolean?`, but any function of one argument is acceptable (though not necessarily sensible). These predicates are used during dispatch to find the proper overload.

Following the class name and list of predicates is a list of method bindings for the class operations. The first component, *method-name* specifies the name of an operation from the class definition. The method binding, *expr*, should evaluate to a function that is compatible with the operation specification from the class definition. The expressions that define instance methods become suspended: the entire expression will be evaluated for each call to the method, therefore any side-effects of the expression will be repeated at each point of instantiation. Because this behavior differs from that for traditional scheme definitions, the expression that defines an instance method should simply be a `lambda` form or a variable. An instance declaration must have a method definition for each class operation that has no default.

The following code shows an instance of the above `Eq` class for integers:

```
(define-instance (Eq integer?)
  (== =))
```

Following the above definition, applying `==` to integers will dispatch to the standard `=` function. However, the class could be redefined in a controlled context using `let-instance` as follows:

```
(let-instance ([(Eq integer?)
                (== eq?)])
  ...)
```

Applications of `==` to integers in the `let-instance` form body will dispatch to the standard `eq?` function.

Class operations are not always open to additional overloads in this system. As shown later, they are implemented as identifier macros (also called symbolic macros), and when referenced expand to an *instantiation*. When a class operation is instantiated, the result is a function that may dispatch only to overloads of the operation that are defined visible at the point of instantiation. In particular, if a function definition calls a class operation, those calls will recognize no new lexical instance declarations introduced before the function itself is called. Continuing the `Eq` class example, consider the following program:

```
(define-instance (Eq char?) (== char=?))

(define elem
  (lambda (m ls)
    (cond
      [(null? ls) #f]
      [(== m (car ls)) #t]
      [else (elem m (cdr ls))])))

(let-instance ([(Eq char?) (== char-ci=?)])
  (elem #\x (list #\X #\Y #\Z)))
```

First an instance of `Eq` is defined for character types, using `char=?`. Next, the `elem` function is implemented. This function is analogous to the Haskell function from Figure 1. The `elem` function implements the same functionality as its Haskell counterpart, but due to the instantiation model of instance methods, calls to the function will dispatch based on the instances visible at the point that `elem` is defined. Thus, even though the next expression shadows the instance declaration for characters, using `char-ci=?` to implement `==`, the call to `elem` still dispatches to the first instance declaration, which uses the case-sensitive comparator, and the expression yields the result `#f`. Had the new instance been defined using `define-instance`, then `elem` would have used the case-insensitive comparator, and the above expression would have yielded `#t`.

4.3 Qualified Functions

The previous example illustrates how class operators and `let-instance` expressions preserve lexical scoping. Unfortunately, this introduces a difference between generic functions implemented as class operators and generic functions that are implemented as Scheme functions that apply class operators. It is beneficial to also have generic Scheme functions implemented in terms of class operators, that exhibit the same overloading behavior as class operators.

Haskell functions are overloaded by expressing their implementations in terms of class operators. When overloaded, a function type is then qualified with the type classes that define the operations used in the function body. Recall the `elem` function defined in Figure 1. It has qualified type `(Eq a) => a -> [a] -> Bool`, which expresses its use of type class operators.

Scheme functions require no such qualification to call class operators, but we borrow the notion to express our more dynamic generic functions, which we call `qualified functions`. Qualified functions take one of the following forms:

```
(define-qualified fn-name (class-name ...)
  expr)

(define-open-qualified fn-name
  (class-name ...)
  expr)
```

The function *expr* defined by this form is qualified by the list of classes (*class-name*...). Qualified functions have the same overload model as class operators. When referenced inside a `let-instance` form that overloads one of the qualifying classes, a qualified function's body can use the lexically introduced overloads. Qualified functions are also subject to instantiation. Inside a qualified function, the operations from the list of qualifying classes dispatch to the overloads visible at the point in the program where the function is *referenced*, rather than the point where the function was *defined*. As such, the behavior of the function can be overloaded at or around call sites. Furthermore, the expression that defines a qualified function is suspended in the same manner as for instance methods. It is thus expected that qualified functions will be implemented with `lambda` forms. However, qualified functions suffer this strange evaluation property in exchange for the ability to dynamically overload their behavior.

Revisiting the `elem` example from the previous section, the function is now defined using `define-qualified`:

```
(define-qualified elem (Eq)
  (lambda (m ls)
    (cond
      [(null? ls) #f]
      [(== m (car ls)) #t]
      [else (elem m (cdr ls))])))
```

The call to `==` within the function body will now dispatch based on the instances visible at the point that `elem` is called, rather than where it was defined. Using this definition of `elem`, the expression:

```
(let-instance ([(Eq char?) (== char-ci=?)])
  (elem #\x (list #\X #\Y #\Z)))
```

yields the value `#t`.

The following program illustrates a qualified function called in two different instance contexts:

```
(define-qualified ==-3 (Eq)
 (lambda (x y z) (and (== x y) (== y z))))

 (cons (let-instance ([(Eq char?)
                       (== char-ci=?)])
         (==-3 #\x #\X #\z))
       (let-instance ([(Eq char?)
                       (== (lambda (a b)
                             #t))])
         (==-3 #\x #\X #\z)))
```

The `==-3` qualified function performs a 3-way equality comparison. Both applications of `==-3` take the same arguments, but each application occurs within the scope of a different instance declaration. This results in dispatch to two different implementations of the `==` method inside the body of the qualified function: the first performing a case-insensitive comparison and the second always yielding `#t`. Evaluation of this expression yields the pair `'(#f . #t)`.

A self-reference inside the body of a function defined with `define-qualified` refers to the current instantiation of the function. However, if a function is defined with `define-open-qualified`, then a self-reference results in a new instantiation of the qualified function. Thus it is possible for such a qualified function to call itself with new instances in scope, as in the following (admittedly bizarre) example:

```
(let-class ([(B p) (o p)])
  (let-instance ([(B boolean?)
                  (o (lambda (x)
                       'boolean))])
    (define-open-qualified f (B)
      (lambda (x)
        (cons
         (o x)
         (if x
             (let-instance ([(B boolean?)
                             (o (lambda (x)
                                  x))])
               (f #f))
             '()))))
    (f #t)))
```

The above expression defines a class, `B`, that specifies one operation of one argument. It then establishes an instance of the class for booleans and defines a function `f` that is qualified over instances of the class. Calling `f` with the value `#t` results in a call to the instance method in the scope of only the outer `let-instance` definition. The result of this call, the symbol `'boolean`, is paired with the result of recurring on `f`, this time in the scope of an instance that implements the instance method `o` as the identity. The final result of these gymnastics is the list `'(boolean #f)`. Whether this functionality serves a useful purpose is a subject of future investigation.

5. Examples

Under some circumstances, a set of instance methods will be implemented such that each applies its own associated class operator. This makes sense especially when defining class instances for data structures that contain values for which instances of the same class exist. For instance, consider the following implementation of `Eq` for Scheme lists:

```
(define-instance (Eq list?)
  [== (lambda (a b)
        (cond
          [(and (null? a) (null? b)) #t]
          [(or (null? a) (null? b)) #f]
          [else (and (== (car a) (car b))
                     (== (cdr a) (cdr b)))]))])
```

This instance of `Eq` requires that `==` be overloaded for every element of the list. The nested calls to `==` in the Scheme implementation are resolved at runtime and will fail if the arguments are not members of the `Eq` class.

Scheme lists result simply from disciplined use of pairs and the null object (`'()`). As such, a more fitting implementation of equality would handle pairs and the null object separately, as in the following:

```
(define-instance (Eq null?)
  [== (lambda (a b) (eq? a b))])
(define-instance (Eq pair?)
  [== (lambda (a b)
        (and (== (car a) (car b))
             (== (cdr a) (cdr b))))])
```

Scheme programs often use lists as their primary data structure, and operate upon them with higher order functions, especially the standard `map` function. Nonetheless, lists are only one data structure among others, trees for instance, and it may be desirable to map a function over other such data structures. The Haskell standard library specifies an overloaded implementation of map, called `fmap`, which varies its implementation depending on the data structure over which it maps. Haskell supports overloading on *type constructors* [Jon93], and this functionality is used to implement generalized mapping.

In Haskell, the `fmap` function is the sole operator of the `Functor` constructor class, which is defined as follows:

```
class  Functor f  where
  fmap :: (a -> b) -> f a -> f b
```

The proper implementation of `fmap` for lists is the standard `map` function, and the instance for lists is simple:

```
instance Functor [] where
  fmap = map
```

where `[]` is the type constructor for lists.

What follows is a Scheme implementation of `fmap` in the same style as the Haskell version:

```
(define-class (Functor p)
  (fmap fn p))

(define-instance (Functor list?)
  (fmap map))
```

In order to match standard Scheme `map`, `fmap` is not curried. The analogous instance declaration for Scheme lists is shown above. Scheme has no notion of type constructor analogous to that in Haskell. This is especially clear in that Scheme lists are heterogeneous: any given list can contain any Scheme value, regardless of its type. Though Haskell considers type constructor to be distinct from types, Scheme has no such distinction, and a simple predicate, such as `list?` for lists, suffices.

Given the above definition of `Functor`, one might define a tree data type and an overload of `fmap` for it as follows:

```
(define-record tree-branch (left right))
(define-record tree-leaf (item))

(define-instance (Functor tree-branch?)
  (fmap
    (lambda (fn branch)
      (make-tree-branch
```

```
      (fmap fn (tree-branch-left branch))
      (fmap fn
            (tree-branch-right branch))))))


(define-instance (Functor tree-leaf?)
  (fmap
    (lambda (fn leaf)
      (make-tree-leaf
       (fn (tree-leaf-item leaf))))))

(fmap add1 (list 1 2 3))

(fmap (lambda (x) (fmap add1 x))
 (make-tree-branch
  (make-tree-leaf (list 1 2 3))
  (make-tree-leaf (list 4 5 6))))
```

This example uses Chez Scheme's record facility for defining data types. The syntax:

```
(define-record rname (slotn ...))
```

defines a new data type and along with it a constructor *make-rname*, a type predicate *rname?* that returns `#f` for any other scheme type, and accessors of the form *rname-slotn* for each element. Most Scheme implementations supply similar facilities.

First, two data types with which trees can be described, `tree-branch` and `tree-leaf`, are defined. Then for each of these data types an instance of `Functor` is defined. Each instance's implementation of `fmap` constructs a new record from the result of recursively applying `fmap` to its components. Finally, two examples of calls to `fmap` are shown. They yield the expected results: a data structure of the same shape with each number incremented by one.

The Common Lisp Object System (CLOS) [GWB91] is another example of a LISP system that provides support for generic functions and overloading. CLOS is an object-oriented system whose dispatch is primarily based on class identity, but it also supports the overloading of generic functions on based on specific values. For example, the CLOS method:

```
(defmethod == ((x number) (y (eql 7)))
  'never)
```

defines an overload of the `==` generic function that is called whenever the first argument is a number and the second argument is exactly equal to the number 7.

Since the system described here supports arbitrary predicates, it too can implement such overloads. The following Scheme code mimics the above:

```
(define-class (Eq a b)
  (== a b))
(define is-seven? (lambda (x) (eq? x 7)))
(define-instance (Eq number? is-seven?)
  [== (lambda (x y) 'never)])
```

A new version of the `Eq` class uses two predicate variables in order to establish the two separate overloads. Then an instance of `Eq` is declared using the `number?` predicate and a hand-crafted predicate that checks for equality to `7`.

6. Translating Predicate Classes to Standard Scheme

Since the predicate class facility that we describe here is implemented using Scheme macros, programs that use them correspond directly to traditional Scheme code, the output of macro expansion. In this section, we illustrate how programs written using this system can be understood in terms of the resulting Scheme code.

The system implementation relies on the syntax-case macro expander's controlled variable capture, as well as `define-syntax` macro definitions scoped within function bodies. However, forms like `let-class` and `let-instance` could be similarly implemented in terms of `letrec-syntax`.

A class definition form, `define-class` or `let-class`, introduces two artifacts to the final program. First, an empty class table is created. In this system, a class table is a list of entries, one for each instance of a class. Each entry in the table is a pair of vectors: a vector of predicates, and a vector of instance methods.

The class definition form also introduces a predicate dispatch function for each operation specified. Based on the operation specification, a function is created that searches the class table, trying to find a set of predicates that match the arguments passed to the function.

For example, consider again the `Eq` class:

```
(define-class (Eq a)
  [(== a _) (lambda (l r) (not (/= l r)))]
  [(/= _ a) (lambda (l r) (not (== l r)))])
```

For illustration purposes, the `==` operation dispatches on its first argument but the `/=` operation dispatches based on its second. The code resulting from this form is similar to what is shown in Figure 3.

The class definition introduces a class table, named `Eq-table`, which starts out empty. Next, the default instance methods are defined. Each default becomes the body of a lambda expression that takes a class table in order to implement recursion among the instance methods. Then for each class operation, `==` and `/=`, a dispatch function is introduced. This function is curried, first accepting a class table and then an arbitrary list of arguments. The bodies of the dispatch functions traverse the instance entries in the class table, searching for a match between the predicate `a` and the dispatch argument. Both dispatch functions access the predicate `a` as the first element of the vector of predicates. Since `==` dispatches based on its first argument, `==-dispatch` runs the predicate on `(car args)`, the first argument to the function, but `/=-dispatch` runs the same predicate on `(cadr args)`, its second argument. If the class had more than one predicate, each predicate would be tried on its corresponding argument in an attempt to detect a matching instance. Finally, `==-dispatch` applies to its arguments the first method in the method vector, `op-vec`, whereas `/=` applies the second method. Each instance is passed the current class table in order to properly support recursion among instance methods.

The instance definition forms, `define-instance` and `let-instance`, introduce new methods to the class instance table and ensure that those instances are visible. To do so, an instance definition produces code that updates the class instance table and defines identifier macros for each class operation. These macros, which are not shown for they are implementation details, cause class operations to recognize the new instance. For example, consider the following expression:

```
(let-instance ([(Eq integer?)
                (== =)])
 (cons
   (list (== 5 6) (/= 5 6))
   (list == /=)))
```

This program introduces an instance of the `Eq` class based on the standard `integer?` predicate, assuming the previously described definition of the class. The `=` function is named as the implementation of the `==` operator, and the `/=` is left undefined, thereby relying upon the default method. Within the scope of this instance definition, both `==` and `/=` are called with integer arguments, and the results are collected alongside their instantiations.

```
(define Eq-table '())

(define ==-default
  (lambda (Eq-table)
    (lambda (l r) (not ((/=-dispatch Eq-table) l r)))))

(define /=-default
  (lambda (Eq-table)
    (lambda (l r) (not ((==-dispatch Eq-table) l r)))))

(define ==-dispatch
  (lambda (Eq-table)
    (lambda args
      (letrec ([loop
                (lambda (table)
                  (let ([pred-vec (caar table)]
                        [op-vec (cdar table)])
                    (cond
                     [(null? table) (error "No matching instance...")]
                     [((vector-ref pred-vec 0) (car args))
                      (apply ((vector-ref op-vec 0) Eq-table) args)]
                     [else (loop (cdr table))])))])
        (loop  Eq-table)))))

(define /=-dispatch
  (lambda (Eq-table)
    (lambda args
      (letrec ([loop
                (lambda (table)
                  (let ([pred-vec (caar table)]
                        [op-vec (cdar table)])
                    (cond
                     [(null? table) (error "No matching instance...")]
                     [((vector-ref pred-vec 0) (cadr args))
                      (apply ((vector-ref op-vec 1) Eq-table) args)]
                     [else (loop (cdr table))])))])
        (loop  Eq-table)))))
```

Figure 3. Expansion of the `Eq` class

The following roughly illustrates the expansion of the above expression:

```
(let ([Eq-table
       (cons
        (cons (vector integer?)
              (vector (lambda (Eq-table) =)
                      /=-default))
        Eq-table)])
  (cons
    (list ((==-dispatch Eq-table) 5 6)
          ((/=-dispatch Eq-table) 5 6))
    (list (==-dispatch Eq-table)
          (/=-dispatch Eq-table))))
```

First the above code adds a new entry to the instance table reflecting the structure of the supplied instance. The entry is a `cons` of two vectors. The first contains the `integer?` predicate, or more generally all the predicates needed to describe the instance. The second vector holds the operators, in the same order as specified in `define-class` (instance operators may be specified in any order and they will be appropriately reordered). This entry is then added to the front of the table and bound to a new lexical variable `Eq-table`. As with the default method implementations, the user-supplied implementation of the `==` method becomes the body of a lambda. Since no implementation is provided for `/=`, the default implementation is substituted.

In this new lexical scope, identifier macros for the operators `==` and `/=` are introduced. These macros handle instantiation of class operators when they are referenced. Thus, following all macro expansion, the calls to the operators in the original code are transformed to calls to the dispatch functions, passing along the proper class table, and then applying the result to the intended arguments. As previously mentioned, class operations are not first class entities. Class operations are implemented using identifier macros, so each class operation expands to replace any reference to it with an expression that applies its associated dispatch function to the class table.

The `define-instance` form differs from its lexical counterpart in that it updates the class table in place. For example, the instance illustrated above could also be written as follows:

```
(define-instance (Eq integer?)
                 (== =))
```

And its expansion is as follows:

```
(set! Eq-table
      (cons
       (cons (vector integer?)
             (vector (lambda (Eq-table) =)
                     /=-default))
            Eq-table))
```

Rather than lexically binding a new table to extend the old one, it applies side effects to the existing table to add the new instance entry.

The `define-qualified` form introduces functions that look up class operation overloads visible at the point where the function is referenced, rather than where the function is defined. To implement

such functionality, this form introduces an implementation function that takes one class table argument for each class that qualifies it. Consider, for example, the following qualified function:

```
(define-qualified assert-equal (Eq)
  (lambda (a b)
    (if (/= a b)
      (error "Not equal!"))))
```

This function uses whatever instance of `Eq` matches its arguments at its instantiation point to test them for inequality. This program expands to the following:

```
(define assert-equal-impl
  (lambda (Eq-table)
    (letrec
      ([assert-equal
        (lambda (a b)
          (if ((/=-dispatch Eq-table) a b)
              (error "Not equal!")))])
      assert-equal)))
```

The `define-qualified` form generates the above function, which takes a class instance table and uses it to dispatch to the proper implementation of the `/=` method, as reflected by the call to `/=-dispatch`. The body of `assert-equal` is wrapped within a `letrec` form and bound to the name `assert-equal` so that self-references refer to the current instantiation. At the top-level, the name `assert-equal` is bound to a macro whose expansion applies the implementation function to the class table. For example, consider the following expression:

```
(cons (assert-equal 5 5)
      assert-equal)
```

Its expansion takes the following form:

```
(cons ((assert-equal-impl Eq-table) 5 5)
      (assert-equal-impl Eq-table))
```

The references to `assert-equal` expand to apply the implementation function, `assert-equal-impl`, to the newly extended class table.

The `assert-equal` qualified function can be implemented using the `define-open-qualified` form as follows:

```
(define-open-qualified assert-equal (Eq)
  (lambda (a b)
    (if (/= a b)
        (error "Not equal!"))))
```

Then only the expansion of the implementation function differs, as shown in the following:

```
(define assert-equal-impl
  (lambda (Eq-table)
    (lambda (a b)
      (if ((/=-dispatch Eq-table) a b)
          (error "Not equal!")))))
```

In this case, the body of the function is no longer wrapped within a `letrec` form. Thus, calls to `asset-equal` within the body of the function refer to the aforementioned macro and are expanded as described above.

7. Related Work

Although type classes in particular have been studied in the statically typed functional programming languages, overloading in general has also been added to dynamically typed programming languages.As mentioned earlier, for example, the Common Lisp Object System (CLOS) provides many of the benefits of an object-oriented programming language. Its design differs from other object-oriented languages in that operations are implemented using *generic functions* in the form of overloaded methods. These methods differ from the methods of most object-oriented languages in that they are not represented as messages passed to an object. Rather they are applied like Lisp functions, but each generic function name can refer to multiple method definitions, each supplied with a different set of *parameter specializers*. This mechanism applies to more than user-defined Lisp classes. Lisp methods can also be overloaded based on native Lisp types as well as equality requirements. Furthermore, specialization can be determined based on arbitrary argument positions in a method. As such, some consider the CLOS generic functions to be a generalization of the typical object-oriented style.

The following code illustrates the implementation of generic methods in Common Lisp:

```
(defmethod == ((x number) (y number))
  (= x y))
(defmethod == ((x string) (y string))
  (string-equal x y))
(defmethod != (x y)
  (not (== x y)))
(defmethod == ((x number) (y (eql 7)))
  'never)
```

The `defmethod` special form is the means by which Common Lisp code expresses generic functions. Each call to `defmethod` introduces an overload. The first two lines establish overloads for the `==` function, one for numbers and one for strings. Each indicates its overload by listing the types of its arguments, and uses the appropriate concrete function to implement the overload. Next, a `!=` generic function is implemented with one overload that places no constraints on its arguments. Its body is expressed in terms of the previously defined `==` function. Finally, a curious overload of the `==` function specifies different behavior if its second argument is the number `7`. Given this definition, the expression `(== 7 7)` yields the symbol `'never`.

Although standard Scheme does not specify a mechanism for implementing overloaded functions, rewrites of the CLOS mechanism are available for certain Scheme implementations [Xer, Bar]).

Overloading functions in Scheme has been the subject of previous research. In [Cox97], a language extension for Scheme is described that adds a mechanism for overloading function definitions. The forms `lambda++` and `define++` extend the definition of an existing function, using either user-supplied predicates or an inferred predicate to determine the proper implementation. In this regard it is similar to the Common Lisp Object System. However, it differs from CLOS in that the implementation combines all overloads at compile time and generates a single function with all dispatch functionality inline. Our design is fully implemented within a macro system, whereas this extension requires modifications to the underlying Scheme implementation.

Other programming languages have also investigated models of overloading. Cecil [Cha93] is a prototype based (or classless) object-oriented programming language that features support for multi-methods [Cha92]. It differs from systems like CLOS in that each method overload is considered to be a member of all the objects that determine its dispatch. These methods thus have privileged access to the private fields of those objects. Cecil has a very flexible notion of objects, and since objects, not types, determine dispatch for Cecil multi-methods, it can capture the full capability of CLOS generic functions, including value equality-based parameter specializers. Furthermore, Cecil resolves multi-method calls using a symmetric dispatch algorithm; CLOS uses a linear model, considering the arguments to a call based on their order in the argument list.

MultiJava [CLCM00] is an extension to the Java [GJSB00] programming language that adds support for symmetric multiple dispatch, as used in Cecil. This work emphasizes backward-compatibility with Java, including support for Java's static method overloading mechanism alongside dynamic multi-method dispatch.

Recently, the language F^G [SL05], an extension of the polymorphic typed lambda calculi of Girard and Reynolds [Gir72, Rey74], introduced mechanisms similar to the design described here. It introduces `concept` and `model` expressions, which are analogous to `let-class` and `let-instance`. It also adds a notion of *generic functions*, which are analogous to our qualified functions, as well as closely related to Haskell overloaded functions. Generic functions can have qualified type parameters much like Haskell, but the dispatch to its equivalent of instance operators is also based on the instances visible at the point of a function call. Generic functions do not have a notion of instantiation however: they have first class status and can be called elsewhere yet still exhibit their dynamic properties. The language F^G is statically typed but its type system does not perform type inference. In this language, instances of a class that have overlapping types cannot exist in the same lexical scope. Our system allows them, but recognizes that they may lead to undesirable results. Furthermore, F^G does not have top-level equivalents to `define-class` and `define-instance`.

In [OWW95], an alternative facility for overloading in the context of Hindley/Milner type inference is described. The language, named System O, differs from Haskell type classes in that overloading is based on individual identifiers rather than type classes. A function may then be qualified with a set of identifiers and signatures instead of a set of type class constraints. Compared to Haskell type classes, System O restricts overloading to only occur based on the arguments to a function. Haskell, in contrast, supports overloading on the return type of a function. As a result of System O's restrictions, it has a dynamic semantics that can be used to reason about System O programs apart from type checking. Haskell type class semantics, on the other hand, are intimately tied to the type inference process. Because of this, it is also possible to prove a soundness result with respect to the type of System O programs. Furthermore, every typeable term in a System O program has a principal type that can be recovered by type inferencing (types must be explicitly used to establish overloads however).

System O's dynamic semantics are very similar to those of the system we describe. Overloaded functions are introduced using the form:

```
inst o : s = e in p
```

where `o` is an overloaded identifier, `s` is a polymorphic type, `e` is the body of the overload, and `p` is a System O expression in which the overload is visible. This form is analogous to our `let-instance` form. However, `inst` introduces an overload only on identifier `o`, whereas `let-instance` defines a set of overloaded operators as described by the specified class.

Overload resolution in System O searches the instances lexically for a fitting overload, much like our system does. As such, System O's dynamic semantics allow shadowing of overloads, as our system does, but the type system forbids this: overloads must be unique. System O's overloaded operators are always dispatched based on the type of the first argument to the function. Our system, however, can dispatch based on any argument position, and uses arbitrary predication to select the proper overload. Also, our system can use multiple arguments to determine dispatch. Finally, though System O's dynamic semantics closely match those of our system, it can be still be implemented as a transformation to the more efficient dictionary-passing style that is often used to describe Haskell type classes.

Some Scheme implementations provide the `fluid-let` form, which supports controlled side-effects over some dynamic extent. To understand how `fluid-let` behaves, consider the following program and its result:

```
(let ([x 5])
  (let ([get-x (lambda () x)])
    (cons (fluid-let ([x 4]) (get-x))
          (get-x))))

=> (4 . 5)
```

The code above lexically binds `x` to the value `5`, and binds `get-x` to a function that yields `x`. Then, two calls to `get-x` are combined to form a pair. The first is enclosed within a `fluid-let` form. The `fluid-let` form side-effects `x`, setting its value to `4` for the dynamic extent of its body. The result of the `fluid-let` form is the result of its body, but before yielding its value, `fluid-let` side-effects `x` again, restoring its original value. Thus, the code:

```
(fluid-let ([x 4]) (get-x))
```

is equivalent to the following:

```
(let ([old-x x] [t #f])
  (set! x 4)
  (set! t (get-x))
  (set! x old-x)
  t)
```

The value of x is stored before assigning `4` to it. Then `get-x` is called and its value stored before restoring `x` to its old value. Finally the expression yields the result of `(get-x)`.

This mechanism is in some respects comparable to our predicate class mechanism. For example, consider the following program:

```
(let ([== #f])
  (define is-equal?
    (lambda (a b) (== a b)))
    (fluid-let ([==
                 (lambda (a b)
                   (if (number? a)
                       (= a b)))])
      (is-equal? 5 5))))
```

It binds the lexical variable `==` to a dummy value, `#f`. Then a function `is-equal?` is implemented in terms of `==`. Finally `==` is effected via `fluid-let`, and within its extent, `is-equal?` is called. This entire expression evaluates to `#t`. Compare the above program to the following, which is implemented using predicate classes:

```
(let-class ([(Eq a) (== a _)])
  (define-qualified is-equal? (Eq)
    (lambda (a b) (== a b)))
  (let-instance ([(Eq number?) (== =)])
    (is-equal? 5 5)))
```

It yields the same result as the `fluid-let` example. Here, `==` is introduced using the `let-class` form. Also, `is-equal?` is now implemented as a qualified function. Then `let-instance` replaces the `fluid-let` form. Due to this example's simplicity, the extra machinery of predicate classes exhibits some syntactic overhead, but programs involving more structure and content may be better formulated using type classes than using `fluid-let` to manually implement the same functionality.

8. Conclusion

Predicate classes loosely determine what properties may guide function dispatch. Traditional object-orientation determines dispatch based on one entity involved in a method call: the class to which the method belongs. Some operations, however, require dispatch based on more than the type of one entity. Idioms such as

the Visitor pattern [GHJV95] have been invented to support dispatch based on multiple types in object-oriented languages. Haskell type classes support dispatch based on all the arguments to a function. However, they specifically rely upon the types of function arguments to guide dispatch. Types can encode some sophisticated properties of objects, including relationships between them, but they cannot capture all runtime properties of programs. Common Lisp generic functions also dispatch on the types of arguments, but as shown earlier, they also support predication based on the particular value of an argument. In this regard, some runtime properties of values are available for dispatch. In our system, any Scheme predicate, meaning any function of one argument that might yield `#f`, can be used to define an instance. Thus, any predicate that is writable in the Scheme language can be used to guide dispatch. Predicates may mimic types, as in the standard Scheme predicates like `integer?`, and one may also compare an argument to some constant Scheme value, just as in Common Lisp. As such the mechanism described here can subsume much of the generic function mechanism in Common Lisp. The Common Lisp Object System, however, orders function specializations based on the inheritance hierarchy of any objects passed as arguments as well as their ordering in the function call. This differs from the Scheme system, which matches symmetrically across all predicates but also relies upon the ordering of and lexical distance to instance definitions. Thus, one may mimic this behavior, but such simulation depends upon the ordering of instance definitions.

The structure imposed by predicate classes provides a means to capture relationships between operations. Related functionality can be described as a unit using a class and subsequently implemented as instances of that class. Applications can thus use predicate classes to organize problem domain abstractions systematically and render them in program text. Such is the organizational power commonly associated with object-orientation; however, CLOS implements an object-oriented system that places less emphasis on the discipline of grouping functionality, preferring to focus on the expressiveness of generic function dispatch. The predicate class mechanism expresses the organization of objects but retains the emphasis on functions, rather than objects, generally associated with functional programming.

The flexibility of dynamic typing must, however, be weighed against the limitations imposed by a lack of static information during compilation. A static type system imposes some limitations on how programs can be written, but this rigidity in turn yields the ability for the language implementation to infer more properties from programs and use this extra information to increase expressiveness. For example, consider the following sketch of the standard Haskell `Num` type class:

```
class Num a where
  ...
  fromInteger :: Integer -> a
  ...
```

The `Num` type class has operations whose arguments do not contain enough information to determine how dispatch will proceed. Specifically, the `fromInteger` method, when applied to an `Integer` value, yields a value of type `a`, where `a` is the overload type. Since this method always takes only an integer, it relies on the return type to distinguish overloads, a feature that our system does not support. In order to implement something like the above in Scheme, the operations must take an additional argument that determines dispatch:

```
(define-class (Num a)
   ...
  (fromInteger a i)
  ...)
```

Here, the `fromInteger` method has an extra parameter, which must be the type to which the supplied integer is converted. Such a contortion is significantly less expressive than the Haskell analogue: a value of the proper type must be available in order to convert another `Integer` to it. This value's sole purpose is to guide dispatch. The change gives the `Num` class an object-oriented feel and muddies the abstraction with implementation details.

In the case of multiple parameter classes, operations need not be dependent upon all the class predicates. Despite interest in and known uses for multiple parameter type classes for Haskell, as well as support for them in several implementations, type checking of programs that make use of them is undecidable in the general case. Nonetheless they are considered useful, and various means have been proposed to make them tractable [Jon00, PJM97, DO02, CKPM05]. In the Scheme system, lack of dispatch information can also be problematic, especially if multiple instances of the class have overlapping predicates. A call to an operation with this sort of ambiguity results in the most recent instance's operation being called. An implementation of this system could recognize such ambiguities and report them as warnings at compile-time and as errors at runtime, but the system we describe here does not.

In Haskell, a class instance method can be overloaded for some other class. In this manner, even a particular method can utilize ad-hoc polymorphism in its implementation. Since methods in the Scheme system are implemented using macros, it is not possible to implement an instance method as a qualified function. One may use such a function as a class method, but it will be bound to a particular class table at the point of its definition so it will not be dynamic over its class qualifications.

As with Haskell, classes that qualify a function must not have overlapping operation names. However, multiple classes whose operation names overlap can be defined, but the behavior for this situation is rather idiosyncratic. Suppose two predicate classes share an operation name. Then at any point in the program, the method name corresponds to the class with the instance that is most recently defined (at the top level using `define-instance`) or most closely defined (using `let-instance`). Thus, instance definitions introduce, or re-introduce, their method names and in doing so shadow the value most recently associated with those names. One may still use commonly-named methods from multiple classes, but this requires the lexical capture of one class's instance method prior to defining an instance of the other class.

Haskell type classes model more of the functionality typical of object-oriented mechanisms than the described Scheme system. For example, type classes can derive from other type classes, much as an object-oriented class can be derived from another using inheritance. The predicate class mechanism does not support the derivation of one type class from another, but this functionality could be added to the system.

Combining the top-level class and instance definitions of Haskell with lexically scoped class and instance definitions increases expressive power. The ability to override an instance declaration as needed lends flexibility to how applications are designed. For example, an application may establish some problem-specific abstractions using classes and provide some default instances for them to handle the common cases. Nonetheless, any portion of the application that uses instance methods or qualified functions may now override the default instances in a controlled fashion as needed. Haskell could also benefit from this capability, though we are unaware of any investigation of such functionality for Haskell.

9. Acknowledgments

This work benefited from discussions with R. Kent Dybvig, Jeremy Siek, and Abdulaziz Ghuloum as well as collaborations with the latter two.

References

[Bar] Eli Barzilay. *Swindle*. http://www.barzilay.org/Swindle/.

[Cha92] Craig Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56, 1992.

[Cha93] C. Chambers. The Cecil language: Specification and rationale. Technical Report TR-93-03-05, 1993.

[CHO92] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes (extended abstract). In *1992 ACM Conference on Lisp and Functional Programming*, pages 170–181. ACM, ACM, August 1992.

[CKPM05] Manuel M. T. Chakravarty, Gabrielle Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13, New York, NY, USA, 2005. ACM Press.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000. ACM Press.

[Cox97] Anthony Cox. Simulated overloading using generic functions in Scheme. Master's thesis, University of Waterloo, 1997.

[DHB92] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, dec 1992.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[DO02] Dominic Duggan and John Ophel. Type-checking multi-parameter type classes. *J. Funct. Program.*, 12(2):133–158, 2002.

[Dyb92] R. Kent Dybvig. Writing hygienic macros in Scheme with syntax-case. Computer Science Department Technical Report #356, Indiana University, Bloomington, Indiana, June 1992.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[Gir72] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thse de doctorat d'état, Université Paris VII, Paris, France, 1972.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[GWB91] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. CLOS: integrating object-oriented and functional programming. *Commun. ACM*, 34(9):29–38, 1991.

[Jon93] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 52–61, New York, NY, USA, 1993. ACM Press.

[Jon00] M. P. Jones. Type classes with functional dependencies. In *Proc. 9th European Symp. on Prog. (ESOP 2000)*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244, New York, NY, March 2000. Springer-Verlag.

[KFFD86] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161, New York, NY, USA, 1986. ACM Press.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[NT02] Matthias Neubauer and Peter Thiemann. Type classes with more higher-order polymorphism. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 179–190, New York, NY, USA, 2002. ACM Press.

[OWW95] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 135–146, New York, NY, USA, 1995. ACM Press.

[Pey03] Simon Peyton Jones. The Haskell 98 language. *Journal of Functional Programming*, 13:103–124, January 2003.

[PJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: Exploring the design space. In *Proceedings of the 1997 Haskell Workshop*, June 1997.

[Rey74] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, New York, NY, 1974. Springer-Verlag.

[SL05] Jeremy Siek and Andrew Lumsdaine. Essential language support for generic programming. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 73–84, New York, NY, USA, 2005. ACM Press.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM Press.

[Xer] Xerox PARC. *Tiny-CLOS*. ftp://ftp.parc.xerox.com/pub/mops/tiny/.