Scheme with Classes, Mixins, and Traits

Matthew Flatt[1], Robert Bruce Findler[2], and Matthias Felleisen[3]

[1] University of Utah
[2] University of Chicago
[3] Northeastern University

Abstract. The Scheme language report advocates language design as the composition of a small set of orthogonal constructs, instead of a large accumulation of features. In this paper, we demonstrate how such a design scales with the addition of a class system to Scheme. Specifically, the PLT Scheme class system is a collection of orthogonal linguistic constructs for creating classes in arbitrary lexical scopes and for manipulating them as first-class values. Due to the smooth integration of classes and the core language, programmers can express mixins and traits, two major recent innovations in the object-oriented world. The class system is implemented as a macro in terms of procedures and a record-type generator; the mixin and trait patterns, in turn, are naturally codified as macros over the class system.

1 Growing a Language

The Revised[5] Report on the Scheme programming language [20] starts with the famous proclamation that "[p]rogramming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary." As a result, Scheme's core expression language consists of just six constructs: variables, constants, conditionals, assignments, procedures, and function applications. Its remaining constructs implement variable definitions and a few different forms of procedure parameter specifications. Everything else is defined as a function or macro.

PLT Scheme [25], a Scheme implementation intended for language experimentation, takes this maxim to the limit. It extends the core of Scheme with a few constructs, such as modules and generative structure definitions, and provides a highly expressive macro system. Over the past ten years, we have used this basis to conduct many language design experiments, including the development of an expressive and practical class system. We have designed and implemented four variants of the class system, and we have re-implemented DrScheme [13]—a substantial application of close to 200,000 lines of PLT Scheme code—in terms of this class system as many times.

Classes in PLT Scheme are first-class values, and the class system's scoping rules are consistent with Scheme's lexical scope and single namespace. Furthermore, the class system serves as a foundation for further macro-based explorations into class-like mechanisms, such as mixins and traits.

A mixin [11] is a class declaration parameterized over its superclass using `lambda`. Years of experience with these mixins shows that they are practical. Scoping rules for

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 270–289, 2006.

methods allow both flexibility and control in combining mixins, while explicit inheritance specifications ensure that unintentional collisions are flagged early.

In this setting, a trait [29] is a set of mixins. Although mixins and traits both represent extensions to a class, we distinguish traits from mixins, because traits provide fine-grained control over individual methods in the extension, unlike mixins.

Last but not least, objects instantiated by the class system are efficient in space and time, whether the class is written directly or instantiated through mixins and or traits. In particular, objects in our system consume a similar amount of space to a Smalltalk or Java object. Method calls have a cost similar to Smalltalk method calls or interface-based Java calls. In short, the class system is efficient as well as effective.

2 Classes

In PLT Scheme, a `class` expression denotes a first-class value, just like a `lambda` expression:

`(class` *superclass-expr decl-or-expr** `)`

The *superclass-expr* determines the superclass for the new class. Each *decl-or-expr* is either a declaration related to methods, fields, and intialization arguments, or it is an expression that is evaluated each time that the class is instantiated. In other words, instead of a method-like constructor, a class has initialization expressions interleaved with field and method declarations. Figure 1 displays a simplified grammar for *decl-or-expr*.

By convention, class names end with `%`. The built-in root class is `object%`. Thus the following expression creates a class with public methods *`get-size`*, *`grow`*, and `eat`:

```
(class object%
  (init size)                ; initialization argument
  (define current-size size) ; field
  (super-new)                ; superclass initialization
  (define/public (get-size)
    current-size)
  (define/public (grow amt)
    (set! current-size (+ amt current-size)))
  (define/public (eat other-fish)
    (grow (send other-fish get-size))))
```

The *`size`* initialization argument must be supplied via a named argument when instantiating the class through the `new` form:

```
(new (class object% (init size) ...) [size 10])
```

Of course, we can also name the class and its instance:

```
(define fish% (class object% (init size) ...))
(define charlie (new fish% [size 10]))
```

In the definition of *`fish%`*, *`current-size`* is a private field that starts out with the value of the *`size`* initialization argument. Initialization arguments like *`size`* are available