First-Class Modules for Perl

Richard Tibbetts and Christopher Lesniewski-Laas

May 16, 2002

Abstract

ML's functors allow developers to write code which generates software modules at compile time. We present a system for a similar, but more powerful, feature for the popular industry-grade language Perl, which is frequently used for implementing Internet services and "glue" software. This enables the modular development of parameterized packages and classes, as well as several other features often implemented, inflexibly, as language syntax. By using Perl's introspection features, we were able to implement our functors without modifying the core Perl language or its interpreter.

1 Background and Motivation

The ability to programmatically generate software modules has long been present in research languages. For example, in ML, a *functor* is a function which maps modules to modules. This feature is frequently in demand, but languages with substantial industry penetration rarely provide adequate support for it. Instead, *ad hoc* solutions using macros and code generators are the rule. Unfortunately, code produced in this way is difficult to debug and maintain: there is no structure imposed on the generated code, and the association between the code generator and clients of the generated code is usually lost.

Perl, a dynamically-typed, object-oriented language, is pragmatic in attitude and very popular, especially for Web and server applications; it has been called "the duct-tape of the Internet"[PFF00]. Although it is optimized for utility rather than elegance, it has some features, such as closures, not found in most industrial languages.

The "package" system built in to Perl is fairly similar to ML's module system: Perl packages contain a set of elements, and can be imported by other packages. However, in Perl, packages are not themselves objects. On the other hand, in ML, packages are "second-class" objects: they constitute the data for a "little language" built on top of ML, but cannot be manipulated by normal ML functions. Functors, the functions of this little language, are completely evaluated at compile time. Thus they are adequate for many applications, such as parameterized modules, but do not support run-time generation or loading of components.

Our desire for parameterized packages in Perl was motivated by a number of real-world problems. Because Perl's object system is built on top of its package system — each class is defined by a package — parameterized packages enable parameterized classes. Auto-generation of class types would be useful in designing data marshalling libraries (desired by one of the authors for an RPC system), and would reduce boilerplate-code duplication for “glue” interfaces (desired by the other author for a database application). We saw functors as a way to solve our problems.

2 Summary

Perhaps unfortunately, integrating parameters into Perl’s native package-definition syntax would have required modifying the interpreter. We elected not to go this route, because it would dramatically reduce the probability that anyone would ever use our software. However, since Perl allows nigh-arbitrary manipulation of its symbol table from within the language, and also has fairly sophisticated syntax-overloading features, we were able to get most of the expressive power that we wanted without requiring any changes to the language’s fundamental definition.

To allow package manipulations to be expressed within the language, we designed a `Package` class to wrap the concept of a package in a run-time structure. The constructor for `Package` generally takes no arguments, and creates a new empty package with a generated name. Alternatively, it can be passed the name of an existing package, in which case it will provide an interface for manipulating that package.

A functor is a standard Perl subroutine that creates a `Package`, inserts the appropriate elements, and returns the name of the package. Because it is a standard Perl subroutine, a functor can take any kind of argument, and can be called at any point during the execution of the program. This increases the range of problems that can be solved using functors. Also, because `Package` is a first-class data type, `Package`s can be memoized, stored as elements of a package or collection, and manipulated in many other ways.

```
 1:  package Foo;
 2:  sub my_print { print "FOO @_\n"; }

 3:  package Bar;
 4:  sub my_print { print "BAR @_\n"; }

 5:  package main;
 6:  sub my_functor {
 7:    my ($arg1, $arg2) = @_;
 8:    my $package = Package->new();
 9:    $package->( "new" ) = sub {
10:      bless [] => "$package";
11:    };
12:    $package->( "method" ) = sub {
13:      $arg1->my_print($arg2);
14:    };
15:    return "$package";
16:  }

17:  my $FooP = my_functor(Foo,123);
18:  my $BarP = my_functor(Bar,456);

17:  my $f = $FooP->new();
18:  my $b = $BarP->new();

21:  $f->method();
22:  $b->method();
```

Figure 1: Example of a Perl functor.

An illustrative example of a simple functor is given in Figure 1. Lines 1-4 define two packages, `Foo` and `Bar`, each with a simple static method that prints its arguments. These packages are not classes, because they have no `new` method. Line 5 switches back to the `main` package. Line

6 begins our functor, named `my_functor`, which takes 2 arguments, a class name and a message. Line 8 creates an anonymous `Package`. Lines 9-11 add a `new` method to the package, making it a class. The `new` method `bless`s a reference to an anonymous array into the generated package, creating a Perl object. Lines 12-14 create a method on the new class, which calls the `my_print` method of `$arg1` on the other functor argument (`$arg2`). Line 15 returns the stringified `Package` object, which will be the name that the package can be manipulated by, its name in the symbol table. Lines 17 and 18 execute the functor on two sets of arguments, creating new packages. Lines 19 and 20 create instances of the generated classes. Lines 21 and 22 show calling the methods on these objects, which successfully calls the methods of `Foo` and `Bar` respectively. The output of this program is

```
FOO Foo 123
BOO Bar 456
```

3 Problems

There are a few problems associated with this implementation; the most important is related to the way subroutine elements are added to a `Package`. Perl subroutines are lexically scoped, and when subroutines are added to a package in the traditional way, their bodies are compiled within the scope of the package. However, when subroutines are added to a `Package`, their bodies are compiled within the scope of the enclosing code, which is generally contained in a different package than that represented by the `Package`. Among other things, this means that they cannot make unqualified references to subroutines and other elements in the generated `Package`.

In a similar way, many types of Perl hackery that depend on proper package scoping don't work straightforwardly with packages produced by `Package`→`new()`. This includes some common idioms, such as importing symbols via the `use` keyword, calling into a superclass via the `SUPER` pseudo-package, and defining fall-back methods via the `AUTOLOAD` method.

However, in general, parameterized packages work. In particular, when used with the standard object-oriented Perl discipline (e.g., `bless`, `@ISA`, `$object`→`method()`), generated packages are equivalent to statically defined packages. Additionally, the ability to generate the symbol table largely obviates the need for some features, such as `AUTOLOAD`. Those other missing features which remain important can be worked around without any real difficulties. In short, there aren't any major Perl features which are unavailable to generated packages, as long as the programmer is aware of the few idioms that need to be specially handled.

4 Evaluation

In general, Perl functors are superior to ML functors, due to their greater flexibility: Perl functors can use and be manipulated by the full power of the Perl language, while ML functors are constrained to their own "little language." This additional power of Perl functors allows them to be more easily integrated into the flow of code. And, for syntactic convenience, they can be passed arbitrary values without having to encapsulate the values into modules.

However, it should be kept in mind that, as in the rest of Perl, module type-checking is done at run-time and is relatively forgiving. Perl generally assumes that the programmer knows what he is doing, rather than forcing the programmer to make type conversions explicit. This might confuse programmers familiar with the very strict type checking of ML. On the other hand, ML programmers aren't our target audience: rather, we're attempting to provide a new abstraction to programmers already familiar with Perl.

The `Package` package is really just a wrapper around symbol table manipulations. Thus, it can be said that, for a sufficiently capable Perl hacker, our functors do not provide additional functionality. However, we feel that functors provide a useful abstraction, and modules written using them are more readable when compared to software written using the equivalent raw symbol table manipulations. They also factor out common kinds of manipulation, and make them more accessible to the average module programmer.

5 Lessons Learned

In implementing Perl functors, we experimented and worked with many of the more obscure features of the Perl interpreter. These aspects of the language, such as the exposed symbol table, exist to provide functionality such as our functors. However, we found insufficient freedom to implement properly everything we would have liked. In particular, we would have preferred to match the scoping of our functors more closely to the scoping of normal packages, but were prevented from doing this by limitations of the Perl parser. This indicates that Perl has failed to provide quite enough flexibility for our application.

To be fair, however, it is a difficult language design problem to leave in the appropriate amount of flexibility in the appropriate places. The fundamental success of this project showed that a minor failure in this aspect of language design is not catastrophic, but careful attention to these details would make for a better language. Since the benefits of implementing language features like functors within the language, rather than extending the language itself, are significant, meta-language extensibility should be an important factor in language design.

References

[PFF00] Perl fast facts. On the web at http://www.perl.org/press/fast_facts.html, October 2000.