How to solve the reuse problem? Functional programming

Philip Wadler
Bell Labs, Lucent Technologies
www.cs.bell-labs.com/~wadler

Much of the reuse supported by functional languages is invisible. For instance, compare the support for lists built into a functional language with the support for lists provided by the C++ Standard Template Library (STL); I elaborate on this below. Other examples include: garbage collection, marshalling (automatic conversion of a datatype to and from a bit stream), trees and algebric data types, and higher-order functions. All of these features save the programmer immense effort, but are not visible as reuse because they are built into the language.

Nonetheless, there is reason to hope that functional languages may provide superior support for reuse in the traditional sense. Factors favoring reuse include: expressiveness of types, lack of side effects (or, conversely, making explicit and available every effect of every operation), and a sophisticated module system.

Built-in lists encapsulate much of the functionality of lists in the STL library, but typically functional programs will be much shorter than the STL counterparts. For instance, the following Haskell expression finds the sum of the squares of the odd numbers from 1 to n.

```
fold (+) 0 [ x*x | x <- [1..n], x `mod` 2 == 1 ]
```

The features exploited here include: enumerators `[1..n]`, list comprehensions `[ term(x) | x <- list, predicate(x) ]`, higher-order functions `(fold (+) 0)`, and lazy evaluation. Lazy evaluation means that lists and iterators over lists are identified. In the above, the list `[1..n]` is manifested lazily, so the space requirement of the program is constant rather than proportional to `n`.

Another factor favoring reuse is that functional languages tend to eschew side effects. The entire semantics is laid bare by the interface to each function, nothing is hidden in global variables. Functional languages typically require that one write more down than in conventional languages, but this is a tradeoff that can ease reuse. For instance, in a functional language, output associated with an operation might be returned as an explicit string (or stream) rather than simply sent to stdout. This may cramp the writer's hand, but it also may ease the reader's brain. When side effects are avoided, the type of the operations makes it explicit that there is output. Further, having the output explicitly in hand makes it easier to process or reroute when required.

Here are three specific examples of reuse in functional programs. Two of the examples are culled from [Wad98], which describes six examples of functional programs used in real-world projects.

1 Reuse in SML/NJ

Some functional languages provide a sophisticated module system, where modules are parameterised by other modules. SML provides such a system; in contrast, Haskell has rather simple-minded modules. The module system of SML encourages the programmer to think carefully about structure, to factor dependencies, and to document the resulting architecture. An especially helpful aspect of modules is sharing declarations. For instance, one may say that a given module requires both a lexer and a symbol table, and use sharing to specify that the identifier type of the lexer is the same as the key type of the symbol table.

The flagship implementation of SML is SML/NJ, a joint project led by Dave MacQueen at Bell Labs and Andrew Appel at Princeton. The SML/NJ compiler produces high-quality code, often comparable in efficiency to C. SML/NJ is itself written in SML and makes extensive use of modules. The module system is extensively used, and everyone involved claims it has contributed greatly to reuse [AM98].

For instance, the register allocation component was written by George Lal at Bell Labs. It makes extensive use of modules and module parameterisation: one module specifies features of the source code, another specifies features of the target architecture. The register allocator has been reused in a number of projects, including an optimising compiler for C. Users report that this was easy to do, thanks in large part to the structuring provided by SML modules.

2 Reuse in Ensemble

Ensemble is a library of protocols that can be used to quickly build distributed applications, making it an example of a reusable framework. Ensemble is in daily use at Cornell to coordinate sharing of keys in a secure network, and to support a distributed CD audio storage and playback service. A number of commercial concerns have begun projects with Ensemble, including BBN, Lockheed Martin, and Microsoft [HvR96].

Ensemble protocol stacks typically have ten or more layers. Highly-layered stacks are flexible, but can be inefficient. Ensemble avoids these inefficiencies by a series of optimisations. The protocol designer segments the code in each the layer, marking common cases. It is a simple matter (currently performed by hand, but easily automated) to trace which segments execute together, and collect these into optimised *trace handlers*. They also cache information to minimise header size and reorder computations to preserve latency. The result is a win-win architecture, offering both modularity and performance.

Ensemble is written entirely in Objective Caml, a dialect of ML. Ensemble beats the performance of its predecessor, Horus, by a wide margin, even though Horus is written in C. To quote the designers, "The use of ML does mean that our current implementation of Ensemble is somewhat slower than it could be, but this has been more than made up for by the ability to rapidly experiment with structural changes, and thereby increase performance through improved design rather than through long hours of hand-coding the entire system in C."

Ensemble demonstrates both the support of reuse by a sophisticated module system, and the invisible reuse promoted by use of a higher-level language.

3 Reuse in Erlang

While most telecommunications firms write phone switches in C++, many Ericsson products are coded in Erlang, a concurrent functional language. Ericsson has a separate division, Erlang Systems, that handles marketing, training, and consulting for Erlang. Over one thousand Ericsson employees have attended Erlang course and over five hundred are currently involved in product development using Erlang. Products written in Erlang contain hundreds of thousands of lines of code, and have earned Ericsson millions of kronor [Arm97].

Erlang is dynamically typed in the same sense as Lisp, Scheme, or Smalltalk, which makes it one of the few modern languages to eschew ML's heritage of static typing. The basic data types are integers (with arbitrary precision, so overflow is not a problem), floats, atoms, tuples, lists, and process identifiers.

Primitives allow one to spawn a process, send a message to a process, or receive a message. Any data value may be sent as a message, and processes may be located on any machine. Erlang uses compression techniques to minimise the bandwidth required to transmit a value. Thus it is both trivial and efficient to send, say, a tree from one machine to another. Compare this with the work required in a language such as C, C++, or Java, where one must separately establish a connection, serialise the tree for transmission, and apply compression. To support robust systems, one process can register to receive a message if another process fails.

Functional programmers often claim that the use of higher-order functions promotes reuse. The classic examples are the *map* and *fold* functions, which encapsulate common forms of list traversal, and just need to be instantiated with an action to perform for each element. Most, but not quite all, list processing can be easily expressed in terms of these functions. The Erlang experience suggests this notion of reuse scales up to support concurrent client-server architectures. A set of libraries encapsulate common server requirements, and just need to be instantiated with the action to be performed for each request. Most, but not quite all, required servers can be easily expressed in terms of these libraries.

References

[Arm97] Joe Armstrong. The development of Erlang. *ACM SIGPLAN International Conference on Functional Programming*, June 1997; *SIGPLAN Notices* 32(8):196–203, August 1997. See: `www.erlang.se`.

[AM98] Andrew Appel, Dave MacQueen, and others. Standard ML of New Jersey. See: `www.cs.bell-labs.com/cm/cs/what/smlnj/`.

[HvR96] Mark Hayden and Robbert vanRenesse. Optimizing Layered Communication Protocols. *Symposium on High Performance Distributed Computing*, Portland, Oregon, August 1997. See: `simon.cs.cornell.edu/Info/Projects/Ensemble`.

[Wad98] Philip Wadler. An angry half dozen. *SIGPLAN Notices* 33(2):25–30, February 1998. [NB. Table of contents on cover of this issue is wrong.]