Open Pattern Matching for C++

Yuriy Solodkyy Gabriel Dos Reis Bjarne Stroustrup

Texas A&M University
College Station, Texas, USA
{yuriys,gdr,bs}@cse.tamu.edu

Abstract

Pattern matching is an abstraction mechanism that can greatly simplify source code. We present functional-style pattern matching for C++ implemented as a library, called *Mach7*[1]. All the patterns are user-definable, can be stored in variables, passed among functions, and allow the use of class hierarchies. As an example, we implement common patterns used in functional languages.

Our approach to pattern matching is based on compile-time composition of pattern objects through concepts. This is superior (in terms of performance and expressiveness) to approaches based on run-time composition of polymorphic pattern objects. In particular, our solution allows mapping functional code based on pattern matching directly into C++ and produces code that is only a few percent slower than hand-optimized C++ code.

The library uses an efficient type switch construct, further extending it to multiple scrutinees and general patterns. We compare the performance of pattern matching to that of double dispatch and open multi-methods in C++.

Categories and Subject Descriptors D.1.5 [*Programming techniques*]: Object-oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design

Keywords Pattern Matching, C++

1. Introduction

Pattern matching is an abstraction mechanism popularized by the functional programming community, most notably ML [12], OCaml [21], and Haskell [15], and recently adopted by several multi-paradigm and object-oriented programming languages such as Scala [30], F# [7], and dialects of C++[22, 29]. The expressive power of pattern matching has been cited as the number one reason for choosing a functional language for a task [6, 25, 28].

This paper presents functional-style pattern matching for C++. To allow experimentation and to be able to use production-quality toolchains (in particular, compilers and optimizers), we implemented our matching facilities as a C++ library.

[1] The library is available at `http://parasol.tamu.edu/mach7/`

GPCE '13, October 27–28, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2517208.2517222

1.1 Summary

We present functional-style pattern matching for C++ built as an ISO C++11 library. Our solution:

- is open to the introduction of new patterns into the library, while not making any assumptions about existing ones.
- is type safe: inappropriate applications of patterns to subjects are compile-time errors.
- Makes patterns first-class citizens in the language (§3.1).
- is non-intrusive, so that it can be retroactively applied to existing types (§3.2).
- provides a unified syntax for various encodings of extensible hierarchical datatypes in C++.
- provides an alternative interpretation of the controversial n+k patterns (in line with that of constructor patterns), leaving the choice of exact semantics to the user (§3.3).
- supports a limited form of views (§3.4).
- generalizes open type switch to multiple scrutinees and enables patterns in case clauses (§3.5).
- demonstrates that compile-time composition of patterns through concepts is superior to run-time composition of patterns through polymorphic interfaces in terms of performance, expressiveness, and static type checking (§4.1).

Our library sets a standard for the performance, extensibility, brevity, clarity, and usefulness of any language solution for pattern matching. It provides full functionality, so we can experiment with the use of pattern matching in C++ and compare it to existing alternatives. Our solution requires only current support of C++11 without any additional tool support.

2. Pattern Matching in C++

The object analyzed through pattern matching is commonly called the *scrutinee* or *subject*, while its static type is commonly called the *subject type*. Consider for example the following definition of factorial in *Mach7*:

```
int factorial(int n) {
  unsigned short m;
  Match(n) {
    Case(0)  return 1;
    Case(m) return m*factorial(m-1);
    Case(_)  throw std::invalid_argument("factorial");
  } EndMatch
}
```

The subject `n` is passed as an argument to the *Match* statement and is then analyzed through *Case* clauses that list various patterns. In the *first-fit* strategy typically adopted by functional languages, the matching proceeds in sequential order while the patterns guarding their respective clauses are *rejected*. Eventually, the statement guarded by the first *accepted* pattern is executed or the control reaches the end of the *Match* statement.