Open Pattern Matching for C++

Yuriy Solodkyy Gabriel Dos Reis Bjarne Stroustrup
Texas A&M University
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 open class hierarchies. As an example, we implement common patterns used in functional languages.

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

Keywords Pattern Matching, C++

1. Introduction

Pattern matching is an abstraction mechanism popularized by the functional programming community, such as ML, OCaml and Haskell, and recently adopted by object-oriented programming languages such as Scala, F#, and dialects of C++ and Java. It provides syntax very close to mathematical notation and allows the user to describe tersely a (possibly infinite) set of values accepted by the *pattern*.

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

- Is *open* to 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.
- Is *non-intrusive* and can be retroactively applied to existing types, both built-in and user-defined.
- Provides a *unified syntax* for dealing with 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.
- Supports a limited form of *views*.
- Generalizes open type switch to *multiple scrutinees* and enables patterns in case clauses.
- 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.

The library sets a standard for performance, extensibility, brevity, clarity and usefulness of any language solution for pattern matching. It provides full functionality, so we can experiment with 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++

As an example of syntax, enabled by *Mach7*, we present here a complete implementation of an evaluator in λ-calculus:

```
struct Term { virtual ~Term() {} };
struct Var : Term { std::string name; };
struct Abs : Term { Var& var; Term& body; };
struct App : Term { Term& func; Term& arg; };

Term* eval(Term* t) {
  var<const Var&> v; var<const Term&> b,a;
  Match(*t) {
    Case(C<Var>())              return &match0;
    Case(C<Abs>())              return &match0;
    Case(C<App>(C<Abs>(v,b),a)) return eval(subs(b,v,a));
    Otherwise() std::cerr << "Invalid term"; return nullptr;
  } EndMatch
}
```

This corresponds directly to a functional language solution involving algebraic data types, which are encoded here with polymorphic classes. The solution is open to class extensions, including those due to dynamic linking, as well as fully supports generic multiple inheritance of C++. The statement also accepts multiple *scrutinees* to allow for re-

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

OOPSLA'13 October 26–31, 2013, Indianapolis, Indiana, USA.

lational checks as demonstrated by the following functional solution to balancing red-black trees:

```
class T { enum color {black,red} col; T* lt; K key; T* rt; };
T* balance(T::color color, T* l, const K& key, T* r) {
  const T::color B = T::color::black, R = T::color::red;
  var⟨T*⟩ a, b, c, d; var⟨K&⟩ x, y, z; T::color col;
  Match(color, l, key, r) {
    Case(B, C⟨T⟩(R, C⟨T⟩(R, a, x, b), y, c), z, d) ...
    Case(B, C⟨T⟩(R, a, x, C⟨T⟩(R, b, y, c)), z, d) ...
    Case(B, a, x, C⟨T⟩(R, C⟨T⟩(R, b, y, c), z, d)) ...
    Case(B, a, x, C⟨T⟩(R, b, y, C⟨T⟩(R, c, z, d))) ...
    Case(col, a, x, b) return new T{col, a, x, b};
  } EndMatch
}
```

The ... in the first four case clauses above stands for:
`return new T{R, new T{B,a,x,b}, y, new T{B,c,z,d}};`.

Built-in types can be analyzed and decomposed through pattern matching too. The following example defines a function for fast computation of Fibonacci numbers with so called "n+k patterns" given here an *equational interpretation* of *application patterns*:

```
int fib(int n) {
  var⟨int⟩ m;
  Match(n) {
    Case(1)       return 1;
    Case(2)       return 1;
    Case(2*m)     return sqr(fib(m+1)) − sqr(fib(m−1));
    Case(2*m+1) return sqr(fib(m+1)) + sqr(fib(m));
  } EndMatch
}
```

The solution is not bound to such equational interpretation. Instead, we offer to look at n+k patterns as mathematical notation that can be used to decompose various mathematical objects into subcomponents in much the same way as constructor patterns decompose algebraic data types.

3. Implementation

Our *Match*-statement extends the efficient type switch for C++ [2]. The core of this extension amounts to handling of multiple subjects (both polymorphic and non-polymorphic) as well as the ability to accept patterns in case clauses.

Our approach to open patterns is based on compile-time composition of pattern objects through *concepts*, using a common C++ technique – *expression templates* [4]. We demonstrate in [3] that it is superior (in terms of performance and expressiveness) to previous approaches based on run-time composition of pattern objects through *interfaces*. In particular, the average overhead of our approach in comparison to a handcrafted solution is 25%, occasionally becoming even faster by about 10%. The average overhead of the approaches based on run-time composition (timed on the same examples) is 934%. None of the approaches introduces a noticeable compile-time overhead.

4. Evaluation Methodology

We performed several independent studies of our approach to demonstrate its effectiveness. The results of most of these studies are shown on the accompanying poster.

The first study compares our approach to the visitor design pattern, while the second does a similar comparison to built-in facilities of Haskell and OCaml. In the third study, we looked at the efficiency of hashing used in type switching on up to four scrutinees on some large real-world class hierarchies. In the fourth study, we compare the performance of matching N polymorphic arguments against double, triple and quadruple dispatch via visitor design pattern as well as open multi-methods extension to C++. In the fifth study we estimate the overhead of our and existing solutions to open patterns in comparison to a handcrafted code, while in the sixth study we compare their impact on the compilation times. In the last study, we rewrote two existing applications in order to compare performance, memory footprint, the ease of use, readability, maintainability etc. of each solution in a real application. The first application was a visitor-based C++ pretty-printer, while the second was a peephole optimizer written in Haskell.

5. Conclusions and Future Work

The *Mach7* library provides functional-style pattern-matching facilities for C++. The solution is open to new patterns, with the traditional patterns implemented as examples. It is non-intrusive and can be applied retroactively. The library provides efficient and expressive matching on multiple scrutinees and compares well to multiple dispatch alternatives in terms of both time and space. We also offer an alternative interpretation of the n+k patterns and show how some traditional generalizations of these patterns can be implemented in our library.

In the future, we would like to implement an actual language extension capable of working with open patterns and look into how code for such patterns can be optimized without hardcoding the knowledge of pattern's semantics into the compiler.

We refer the reader to the original publications [2, 3] as well as first author's PhD thesis [1] for a detailed discussion of our solution to open and efficient pattern matching in C++.

References

[1] Y. Solodkyy. *Simplifying the Analysis of C++ Programs*. PhD thesis, Texas A&M University, August 2013.

[2] Y. Solodkyy, G. Dos Reis, and B. Stroustrup. Open and efficient type switch for C++. In *Proc. ACM OOPSLA'12*, pp 963–982. ACM.

[3] Y. Solodkyy, G. Dos Reis, and B. Stroustrup. Open pattern matching for C++. In *Under review to GPCE'13*.

[4] T. Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995.