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 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, for example 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, 26, 28].

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

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

GPCE'13 October 26–31, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM [to be supplied]. . . $10.00

1.1 Summary

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 (§3.1).
- Is non-intrusive and 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 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 called *subject*, while its static type – *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.

The value 0 in the first case clause is an example of a *value pattern*. It will match only when the subject n is 0. The variable m in the second case clause is an example of a *variable pattern*. It will bind to any value that can be represented by its type. The name _ in the last case clause refers to the common instance of the *wildcard pattern*. Value, variable and wildcard patterns are typically referred to as *primitive patterns*. We extend the list of primitive patterns with a *predicate pattern* whereby we allow the use of any unary predicate or nullary member-predicate as a pattern: e.g. *Case*(even) ... (assuming **bool** even(**int**);) or *Case*([](**int** m) { **return** m^m−1; }) ... for λ-expressions.

A *guard* is a predicate attached to a pattern that may make use of the variables bound in it. The result of its evaluation will determine whether the case clause and the body associated with it will be *accepted* or *rejected*. Combining patterns with guards gives raise to *guard patterns*, which in *Mach7* are expressions of the form $P|{=}E$, where P is a pattern and E is an expression guarding it.

Pattern matching is also closely related to *algebraic data types* – a possibly recursive *sum type* of *product types*. In ML and Haskell, an *Algebraic Data Type* is a data type each of whose values are picked from a disjoint sum of data types, called *variants*. Each variant is a product type, marked with a unique symbolic constant called a *constructor*. Constructors provide a convenient way of creating a value of its variant type as well as discriminating among variants through pattern matching. In particular, given an algebraic data type $D = C_1(T_{11},...,T_{1m_1})|\cdots|C_k(T_{k1},...,T_{km_k})$ an expression of the form $C_i(x_1,...,x_{m_i})$ in a non-pattern-matching context is called a *value constructor* and refers to a value of type D created via constructor C_i and arguments $x_1,...,x_{m_i}$. The same expression in the pattern-matching context is called *constructor pattern* and is used to check whether the subject is of type D and was created with constructor C_i. If so, it binds the actual values it was constructed with to variables $x_1,...,x_{m_i}$ (or matches against nested patterns if x_j are other kinds of patterns).

C++ does not provide a direct support for algebraic data types, however they can be encoded in the language in a number of ways. Common object-oriented encodings employ an abstract class to represent the algebraic data type and derived classes to represent variants. Consider for example the following representation of terms in λ-calculus in C++:

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

C++ allows a class to have several constructors and does not allow overloading the meaning of construction for the use in pattern matching. This is why in *Mach7* we have to be slightly more explicit about constructor patterns, which take the form $\mathsf{C}\langle T_i\rangle(P_1,...,P_{m_i})$, where T_i is the name of the user-defined type we are decomposing and $P_1,...,P_{m_i}$ are patterns that will be matched against members of T_i (§3.2). "C" was chosen to abbreviate "Constructor pattern" or "Case class" as its use resembles the use of case classes in Scala [30]. For example, we can write a complete recursive implementation of an equality of two lambda terms:

```
bool operator==(const Term& left, const Term& right) {
  var⟨const std::string &⟩ s; var⟨const Term&⟩ x,y;
  Match(left             , right          ) {
    Case(C⟨Var⟩(s)      , C⟨Var⟩(+s)      ) return true;
    Case(C⟨Abs⟩(x,y), C⟨Abs⟩(+x,+y)) return true;
    Case(C⟨App⟩(x,y), C⟨App⟩(+x,+y)) return true;
    Otherwise()                             return false ;
  } EndMatch
}
```

This == is an example of a *binary method* – an operation that requires both arguments to have the same type [3]. In each of the case clauses, we check that both subjects are of the same dynamic type using constructor pattern. We then decompose both subjects into components and compare them for equality with a combination of a variable pattern and an equivalence combinator + applied to the same variable pattern. The use of equivalence combinator turns binding use of a variable pattern into a non-binding use of that variable's current value as a value pattern.

In general, *pattern combinator* is an operation on patterns to produce a new pattern. Other typical pattern combinators supported by many languages are *conjunction*, *disjunction* and *negation* combinators with intuitive Boolean interpretation. We add a few non-standard combinators to *Mach7* that reflect the specifics of C++, e.g. the presence of pointers and references in the language.

The equality operator on λ-terms demonstrates nesting of patterns. The variable pattern was nested within an equivalence pattern, which in turn was nested inside a constructor pattern. Nesting allows one to combine many different patterns in many different ways, which makes them particularly expressive. Here is a well-known functional solution to balancing red-black trees with pattern matching due to Chris Okasaki [32, §3.3] implemented in *Mach7*:

```
class T { enum color {black,red} col; T* left; K key; T* right; };
T* balance(T::color clr, T* left, const K& key, T* right) {
  const T::color B = T::black, R = T::red;
  var⟨T*⟩ a, b, c, d; var⟨K&⟩ x, y, z; T::color col;
  Match(clr, left, key, right) {
    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}};.

To demonstrate the openness of the library, we implemented numerous specialized patterns that often appear in practice and are even built into some languages. For example, the following combination of regular-expression and one-of patterns can be used to match against a string and see whether the string represents a toll-free phone number.

```
rex("([0-9]+)-([0-9]+)-([0-9]+)", any({800,888,877}), n, m)
```

The regular-expression pattern takes a C++11 regular expression and an arbitrary number of sub-patterns. It then uses matching groups to match against the sub-patterns. A one-of pattern simply takes an initializer-list with a set of values and checks that the subject matches one of them. The variables n and m are integers, and the values of the last two parts of the pattern will be assigned to them. The parsing is generic and will work with any data type that can be read from an input stream – a common idiom in C++. Should we also need the exact area code, we can mix in a variable pattern with conjunction combinator: a && any(...).

3. Implementation

The traditional object-oriented approach to implementing first-class patterns is based on run-time compositions through interfaces. This "*patterns as objects*" approach has been explored in several different languages [11, 14, 34, 46]. Implementations differ in where bindings are stored and what is returned as a result, but in its most basic form it consists of the pattern interface with a virtual function match that accepts a subject and returns whether it was accepted or rejected. This approach is open to new patterns and pattern combinators, but a mismatch in the type of the subject and the type accepted by the pattern can only be detected at run-time. Furthermore, it implies significant run-time overheads (§4.1).

3.1 Patterns as Expression Templates

Patterns in *Mach7* are represented as objects. They are composed at compile time, based on C++ concepts. *Concept* is the C++ community's long-established term for a set of requirements for template parameters. Concepts were not included in C++11, but techniques for emulating them with enable_if [18] have been in use for a while. In this work, we use the notation for *template constraints* – a simpler version of concepts [41]. The *Mach7* implementation emulates these constraints.

There are two main constraints on which the entire library is built: PATTERN and LAZYEXPRESSION.

```
template ⟨typename P⟩ constexpr bool PATTERN() {
  return COPYABLE⟨P⟩
      && is_pattern⟨P⟩::value
      && requires (typename S, P p, S s) {
           bool = { p(s) };
           AcceptedType⟨P,S⟩;
         };
}
```

The PATTERN constraint is the analog of the pattern interface from the *patterns as objects* solution. Objects of any class P satisfying this constraint are patterns and can be composed with any other patterns in the library as well as be used in the *Match* statement.

Patterns can be passed as arguments of a function, so they must be COPYABLE. Implementation of pattern combinators requires the library to overload certain operators on all the types satisfying the PATTERN constraint. To avoid overloading these operators for types that satisfy the requirements accidentally, PATTERN constraint is a semantic constraint and classes that claim to satisfy it have to state this explicitly by specializing is_pattern⟨P⟩ trait. The constraint introduces also some syntactic requirements, described by the **requires** clause. In particular, patterns require presence of an application operator that serves as an analog of pattern::match(**const** object&) interface method in the *patterns as objects* approach. However, PATTERN does not impose further restrictions on the type of the subject S. Patterns like wildcard pattern will leave the S type completely unrestricted, while other patterns may require it to satisfy certain constraints, model a given concept, inherit from a certain type, etc. Application operator will typically return a value of type **bool** indicating whether the pattern is *accepted* on a given subject (**true**) or *rejected* (**false**).

Most of the patterns are applicable only to subjects of a given *expected type* or types convertible to it. This is the case, for example, with value and variable patterns, where the expected type is the type of the underlying value, as well as with the constructor pattern, where the expected type is the type of the user-defined type it decomposes. Some patterns, however, do not have a single expected type and may work with subjects of many unrelated types. A wildcard pattern, for example, can accept values of any type without involving a conversion. To account for this, the PATTERN constraint requires presence of a type alias AcceptedType, which given a pattern of type P and a subject of type S returns an expected type AcceptedType⟨P,S⟩ that will accept subjects of type S with no or minimum conversions. By default, the alias is defined in terms of a nested type function accepted_type_for as following:

```
template⟨typename P, typename S⟩
  using AcceptedType = P::accepted_type_for⟨S⟩::type;
```

The wildcard pattern defines accepted_type_for to be an identity function, while variable and value patterns define it to be their underlying type. Here is an example of how the variable pattern satisfies the PATTERN constraint:

```
template ⟨REGULAR T⟩ struct var {
  template ⟨typename⟩
    struct accepted_type_for { typedef T type; };
  bool operator()(const T& t) const // exact match
    { m_value = t; return true; }
  template ⟨REGULAR S⟩
  bool operator()(const S& s) const // with conversion
    { m_value = s; return m_value ==s; }
  mutable T m_value; // value bound during matching
};
template ⟨REGULAR T⟩ struct is_pattern⟨var⟨T⟩⟩
  { static const bool value = true; };
```

For semantic or efficiency reasons a pattern may have several overloads of the application operator. In the example, the first alternative is used when no conversion is required and thus the variable pattern is guaranteed to be accepted. The second may involve a possibly narrowing conversion, which is why we check that the values compare equal after assignment. Similarly, for type checking reasons, accepted_type_for may and typically will provide several partial or full specializations to limit the set of acceptable subjects. For example, *address combinator* can only be applied to subjects of pointer types, so its implementation will report a compile-timer error when applied to any non-pointer type.

Guard and n+k patterns, the equivalence combinator, and potentially some new user-defined patterns, depend on capturing the structure (term) of lazily evaluated expressions. All such expressions are objects of some type E that must satisfy the LAZYEXPRESSION constraint:

```
template ⟨typename E⟩ constexpr bool LAZYEXPRESSION() {
  return COPYABLE⟨E⟩
      && is_expression⟨E⟩::value
      && requires (E e) {
           ResultType⟨E⟩;
           ResultType⟨E⟩ =={ eval(e) };
           ResultType⟨E⟩ { e };
         };
}
template⟨typename E⟩ using ResultType = E::result_type;
```

The constraint is again semantic and the classes claiming to satisfy it must assert it through is_expression⟨E⟩ trait. Template alias ResultType⟨E⟩ is defined to return the expression's associated type result_type, which defines the type of the result of a lazily evaluated expression. Any class satisfying the LAZYEXPRESSION constraint must also provide an implementation of function eval that evaluates the result of the expression. Conversion to the result_type should call eval on the object in order to allow the use of lazily evaluated expressions in the contexts where their eagerly computed value is expected: e.g. non-pattern matching context of the right hand side of the *Case* clause. Class var⟨T⟩, for example, models concept LAZYEXPRESSION as following:

```
template ⟨REGULAR T⟩ struct var {
  // ...definitions from before
  typedef T result_type; // type when used in expression
  friend const result_type& eval(const var& v) // eager evaluation
    { return v.m_value; }
  operator result_type() const { return eval(*this); }
};
```

To capture the structure of an expression, the library employs a commonly used technique called “expression templates” [44, 45]. It captures the structure of expression through the type, which for binary addition may look as following:

```
template ⟨LazyExpression E1, LazyExpression E2⟩
struct plus {
  E1 m_e1; E2 m_e2; // subexpressions
  plus(const E1& e1, const E2& e2) : m_e1(e1), m_e2(e2) {}
  typedef decltype(std:: declval ⟨E1::result_type ⟩()
                 + std:: declval ⟨E2::result_type ⟩()
                 ) result_type ; // type of result
  friend result_type eval(const plus& e)
    { return eval(e.m_e1) + eval(e.m_e2); }
  friend plus operator+(const E1& e1, const E2& e2)
    { return plus(e1,e2); }
};
```

The user of the library never sees this definition, instead she implicitly creates its objects with the help of overloaded **operator**+ on any LazyExpression arguments. The type itself models the LazyExpression concept as well so that the lazy expressions can be composed. Notice that all the requirements of the concept are implemented in terms of the requirements on the types of the arguments. The key point to the efficiency of expression templates is that all the types in the final expression are known at compile time, while all the function calls are trivial and fully inlined. Use of new C++11 features like move constructors and perfect forwarding allows us to ensure further that no temporary objects will ever be created at run-time and that the evaluation of the expression template will be as efficient as a hand coded function.

In general, an *expression template* is an algebra $\langle T_C, \{f_1, f_2, ...\}\rangle$ defined over the set $T_C = \{\tau \mid \tau \vDash C\}$ of all the types τ modeling a given concept C. Operations f_i allow one to compose new types modeling concept C out of existing ones. In this sense, the types of all lazy expressions in *Mach7* stem from a set of few possibly parameterized basic types like var⟨T⟩ and value⟨T⟩ (which model LazyExpression) by applying type functors plus, minus ... etc. to them. Every type in the resulting family then has a function eval defined on it that returns a value of the associated type result_type. Similarly, the types of all the patterns stem from a set of few possibly parameterized patterns like wildcard, var⟨T⟩, value⟨T⟩, C⟨T⟩ etc. by applying to them pattern combinators like conjunction, disjunction, equivalence, address etc. The user is allowed to extend both algebras with either basic expressions and patterns or functors and combinators.

Sets $T_{LazyExpression}$ and $T_{Pattern}$ have non-empty intersection, which slightly complicates the matter. Basic types var⟨T⟩ and value⟨T⟩ belong to both families and so do some of the combinators: e.g. conjunction. Since we can only have one overloaded **operator**&& for a given combination of argument types, we have to state conditionally whether requirements of Pattern, LazyExpression or both are satisfied in a given instantiation of conjunction⟨T_1,T_2⟩ depending on what combination of these concepts the argument types T_1 and T_2 model. Concepts, unlike interfaces, allow modeling such behavior without multiplying implementations or introducing dependencies.

3.2 Structural Decomposition

Mach7’s constructor patterns C⟨T⟩(P_1,...,P_n) requires the library to know which member of class T should be used as the subject to P_1, which should be matched against P_2 etc. In functional languages supporting algebraic data types, such decomposition is unambiguous as each variant has only one constructor, which is thus also used as *deconstructor* [2, 13] to define decomposition of a type through pattern matching. In C++, a class may have several constructors, so we must be explicit about a class’ decomposition. We specify that by specializing the library template class bindings. Here are the definitions required to decompose the lambda terms we introduced in §2:

```
template ⟨⟩struct bindings⟨Var⟩ { Members(Var::name); };
template ⟨⟩struct bindings⟨Abs⟩ { Members(Abs::var, Abs::body); };
template ⟨⟩struct bindings⟨App⟩ { Members(App::func, App::arg); };
```

Variadic macro Members simply expands each of its argument into the following definition, demonstrated here on App::func:

```
static decltype(&App::func) member1() { return &App::func; }
```

Each of such functions returns a pointer-to-member that should be bound in position i. The library applies corresponding members to the subject in order to obtain subjects for sub-patterns $P_1, ..., P_n$. The functions get inlined so the code to access a member in a given position becomes the same as the code to access that member directly. Note that binding definitions made this way are *non-intrusive* since the original class definition is not touched. They also respect *encapsulation* since only the public members of the target type will be accessible from within bindings specialization. Members do not have to be data members only, which can be inaccessible, but any of the following three categories:

- Data member of the target type T
- Nullary member function of the target type T
- Unary external function taking the target type T by pointer, reference or value.

Binding definitions have to be written only once for a given class hierarchy and can be used everywhere. This is also true for parameterized classes (e.g., see §3.4). Unfortunately, at this point C++ does not provide sufficient compile-time introspection capabilities to let the library generate these definitions implicitly.

3.3 Algebraic Decomposition

Traditional approaches to generalizing n+k patterns treat matching a pattern $f(x,y)$ against a value v as solving an equation $f(x,y) = v$ [33]. This interpretation is well defined when there are zero or one solutions, but alternative interpretations are possible when there are multiple solutions. Instead of discussing which interpretation is the most general or appropriate, we look at n+k patterns as a *notational decomposition* of mathematical objects. The elements of the notation are associated with sub-components of the matched mathematical entity, which effectively lets us decompose it into parts. The structure of the expression tree used in the notion is an analog of a constructor symbol in structural decomposition, while its leaves are placeholders for parameters to be matched against or inferred from the mathematical object in question. In essence, *algebraic decomposition* is to mathematical objects what structural decomposition is to algebraic data types. While the analogy is somewhat ad-hoc, it resembles the situation with operator overloading: you do not strictly need it, but it is so syntactically convenient it is virtually impossible not to have it. We demonstrate this alternative interpretation of the n+k patterns with examples.

- An expression n/m is often used to decompose a rational number into numerator and denominator.
- Euler notation $a + bi$ with i being an imaginary unit is used to decompose a complex number into real and imaginary parts. Similarly, expressions $r(cos\phi + i\sin\phi)$ and $re^{i\phi}$ are used to decompose it into polar form.
- An object representing 2D line can be decomposed with slope-intercept form $mX + c$, linear equation form $aX + bY = c$ or two-points form $(Y - y_0)(x_1 - x_0) = (y_1 - y_0)(X - x_0)$.
- An object representing a polynomial can be decomposed for a specific degree: a_0, $a_1X^1 + a_0$, $a_2X^2 + a_1X^1 + a_0$ etc.

- An element of a vector space can be decomposed along some sub-spaces of interest. For example a 2D vector can be matched against $(0,0)$, aX, bY, $aX + bY$ to separate the general case from those when one or both components of vector are 0.

Expressions i, X and Y in these examples are not variables, but named constants of some dedicated type that lets the expression be generically decomposed into orthogonal parts. The linear equation and the two-point form for decomposing lines include an equality sign, so it is hard to give them semantics in an equational approach. However, for many interesting cases, the equational approach can be generically expressed in our framework.

The user of our library defines the semantics of decomposing a value of a given type S against an expression of shape E by overloading a function:

```
template ⟨LazyExpressionE, typename S⟩
bool solve(const E&, const S&);
```

The first argument of the function takes an expression template representing a term we are matching against, while the second argument represents the expected result. Note that even though the first argument is passed with const-qualifier, it may still modify state in E. For example, when E is var⟨T⟩, the application operator for const-object that will eventually be called will update a mutable member m_value. The following example defines a generic solver for multiplication by a constant:

```
template ⟨LazyExpressionE, typename T⟩
    requires Field⟨E::result_type⟩()
bool solve(const mult⟨E,value⟨T⟩⟩& e, const E::result_type& r)
    { return solve(e.m_e1,r/eval(e.m_e2)); }
template ⟨LazyExpressionE, typename T⟩
    requires Integral⟨E::result_type⟩()
bool solve(const mult⟨E,value⟨T⟩⟩& e, const E::result_type& r) {
    T t = eval(e.m_e2);
    return r%t ==0 && solve(e.m_e1,r/t);
}
```

The first overload is only applicable when the type of the result of the sub-expression models the Field concept. In this case, we can rely on the presence of a unique inverse and simply call division without any additional checks. The second overload uses integer division, which does not guarantee the unique inverse, and thus we have to verify that the result is divisible by the constant first. This last overload combined with a similar solver for addition of integral types is everything the library needs to express fast power or fibbonaci functions that involve generalized n+k patterns like 2*m+1 and 2*m to distinguish and decompose odd and even numbers respectively.

3.4 Views

Any type T may have an arbitrary number of *binding*s associated with it, which are specified by varying the second parameter of the bindings template – *layout*. The layout is a non-type template parameter of an integral type that has a default value and is thus omitted most of the time. Support of multiple bindings through layouts in our library effectively enables a facility similar to Wadler's *views*[47]. Consider:

```
enum { cartesian = default_layout, polar }; // Layouts
template ⟨class T⟩ struct bindings⟨std::complex⟨T⟩⟩
  { Members(std::real⟨T⟩,std::imag⟨T⟩); };
template ⟨class T⟩ struct bindings⟨std::complex⟨T⟩, polar⟩
  { Members(std::abs⟨T⟩,std::arg⟨T⟩); };
template ⟨class T⟩ using Cart = view⟨std::complex⟨T⟩⟩;
template ⟨class T⟩ using Pole = view⟨std::complex⟨T⟩, polar⟩;
```

```
std::complex⟨double⟩ c; double a,b,r,f;
Match(c)
  Case(Cart⟨double⟩)(a,b)) ...// default layout
  Case(Pole⟨double⟩)(r,f)) ...// view for polar layout
EndMatch
```

The C++ standard effectively enforces the standard library to use Cartesian representation[17, §26.4-4], which is why we choose the Cart layout to be the default. We then define bindings for each layout and introduce template aliases (an analog of typedefs for parameterized classes) for each of the views. *Mach7* class view⟨T,l⟩ binds together a target type with one of its layouts, which can be used everywhere where an original target type was expected.

The important difference from Wadler's solution is that our views can only be used in a match expression and not as a constructor or arguments of a function etc.

3.5 Match Statement

The *Match* statement presented in this paper extends the efficient type switch for C++ [40] to handle multiple subjects (both polymorphic and non-polymorphic) (§3.5.1) and to accept patterns in case clauses (§3.5.2).

3.5.1 Multi-argument Type Switching

The core of our efficient type switch [40] is based on the fact that virtual table pointers (vtbl-pointers) uniquely identify subobjects in the object and are perfect for hashing. The optimal hash function H_{kl}^{V} built for a set of vtbl-pointers V seen by a type switch was chosen by varying parameters k and l to minimize the probability of conflict. Parameter k represented the logarithm of the size of cache, while parameter l represented the number of low bits to ignore.

A *Morton order* (aka *Z-order*) is a function that maps multidimensional data to one dimension while preserving locality of the data points [23]. A Morton number of an N-dimensional coordinate point is obtained by interleaving the binary representations of all coordinates. The original one-dimensional hash function H_{kl}^{V} applied to arguments $v \in V$ was producing hash values in a tight range $[0..2^k[$ where $k \in [K, K+1]$ for $2^{K-1} < |V| \le 2^K$. The produced values were close to each other, which improved the performance of cache due to locality of references. The idea is thus to use Morton order on these hash values and not on the original vtbl-pointers in order to maintain the locality of references. To do this, we still maintain a single parameter k reflecting the size of cache, however we keep N optimal offsets l_i for each argument i.

Consider a set $V^N = \{\langle v_1^1, ..., v_1^N\rangle, ..., \langle v_n^1, ..., v_n^N\rangle\}$ of N-dimensional tuples representing the set of vtbl-pointer combinations coming through a given *Match* statement. As with one-dimensional case, we restrict the size 2^k of the cache to be not larger than twice the closest power of two greater or equal to $n = |V^N|$: i.e. $k \in [K, K+1]$, where $2^{K-1} < |V^N| \le 2^K$. For a given k and offsets $l_1, ..., l_N$ a hash value of a given combination $\langle v^1, ..., v^N\rangle$ is defined as $H_{kl_1...l_N}(\langle v^1, ..., v^N\rangle) = \mu(\frac{v^1}{2^{l_1}}, ..., \frac{v^N}{2^{l_N}}) \mod 2^k$, where function μ returns Morton number (bit interleaving) of N numbers.

Similar to one-dimensional case, we vary parameters $k, l_1, ..., l_N$ in their finite and small domains to obtain an optimal hash function $H_{kl_1...l_N}^{V^N}$ by minimizing the probability of conflict on values from V^N. Unlike the one-dimensional case, we do not try to find the optimal parameters every time we reconfigure the cache. Instead, we only try to improve the parameters to render fewer conflicts in comparison to the number of conflicts rendered by the current configuration. This does not prevent us from eventually converging to the same optimal parameters, which we do over time, but is important for maintaining the amortized complexity of the access constant. We demonstrate in §4.3 that similarly to one-dimensional case such hash function produces little collisions on real-world class hierarchies, while is simple enough to compute to compete with alternatives dealing with multiple dispatch.

3.5.2 Support for Patterns

Given a statement $\mathit{Match}(\mathsf{e}_1,\ldots,\mathsf{e}_N)$ applied to arbitrary expressions e_i, the library introduces several names into the scope of the statement: e.g. number of arguments N, subject types $\mathsf{subject\_type}_i$ (defined as **decltype**(e_i) modulo type qualifiers), number of polymorphic arguments M etc. When $M > 0$ it also introduces the necessary data structures to implement efficient type switching [40]. Only the M arguments whose $\mathsf{subject\_type}_i$ are polymorphic will be used for fast type switching.

For each case clause $\mathit{Case}(\mathsf{p}_1,\ldots,\mathsf{p}_N)$ the library ensures that the number of arguments to the case clause N matches the number of arguments to the *Match* statement, and that the type P_i of every expression p_i passed as its argument models the PATTERN concept. For each $\mathsf{subject\_type}_i$ it then introduces $\mathsf{target\_type}_i$ into the scope of the case clause, defined as the result of evaluating type function P_i::accepted_type_for$\langle\mathsf{subject_type}_i\rangle$. This is the type the pattern expects as an argument on the subject of type $\mathsf{subject\_type}_i$ (§3.1), which is used by the type switching mechanism to properly cast the subject if necessary. The library then introduces names match_i of type $\mathsf{target\_type}_i\&$ bound to properly casted subjects and available to the user in the right-hand side of the case clause in case of a successful match. The qualifiers applied to the type of match_i reflect the qualifiers applied to the type of subject e_i. Finally, the library generates the code that sequentially applies each pattern to properly casted subjects, making the clause's body conditional:

if $(\mathsf{p}_1(\mathsf{match}_1)\ \&\&\ \ldots\&\&\ \mathsf{p}_N(\mathsf{match}_N))$ { /∗ body ∗/ }

When type switching is not involved, the generated code implements the naive backtracking strategy, which is known to be inefficient as it can produce redundant computations [5, §5]. More efficient algorithms for compiling pattern matching have been developed since [1, 21, 24, 25, 37]. Unfortunately, while these algorithms cover most of the typical kinds of patterns, they are not pattern agnostic as they make assumptions about semantics of concrete patterns. A library-based approach to pattern matching is agnostic of the semantics of any given user-defined pattern. The interesting research question in this context would be: what language support is required to be able to optimize open patterns.

The main advantage from using pattern matching in *Mach7* comes from the fast type switching weaved into the *Match* statement. It effectively skips case clauses that will definitely be rejected because their target types are not subtypes of the subjects' dynamic types. This, of course, is only applicable to polymorphic arguments, for non-polymorphic arguments the matching is done naively with cascade of conditional statements.

4. Evaluation

We performed several independent studies of our pattern matching solution to test its efficiency and impact on compilation process. In the first study we compare various functions written with pattern matching and hand-optimized manually in order to estimate the overhead added by the composition of patterns (§4.1). We demonstrate this overhead for both our solution based on compile-time composition of patterns and the suggested alternatives based on the run-time composition of patterns. In the second study we compare a relevant impact on the compilation times brought by both approaches (§4.2). In the third study, we looked at how well our extension of *Match* statement to N-arguments using the Morton order deals with large real-world class hierarchies (§4.3). In the third 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++ (§4.4). In the last study, we rewrote a code optimizer of an experimental language from Haskell into C++ with *Mach7*. We compare the ease of use, readability and maintainability of the original Haskell code and its *Mach7* equivalent (§4.5).

The studies involving performance comparisons have been performed on a Sony VAIO® laptop with Intel® Core™i5 460M CPU at 2.53 GHz; 6GB of RAM; Windows 7 Professional. All the code was compiled with G++ (versions 4.5.2, 4.6.1 and 4.7.2; executed with -O2 to produce x86 binaries under MinGW) as well as Visual C++ (versions 10.0 and 11.0 with profile-guided optimizations).

To improve accuracy, timing was performed using the x86 RDTSC instruction. For every number reported we ran 101 experiments timing 1,000,000 top-level calls each (depending on arguments, there may have been a different number of recursive calls). The first experiment was serving as a warm-up, and typically resulted in an outlier with the largest time. Averaged over 1,000,000 calls, the number of cycles per top-level call in each of the 101 experiments was sorted and the median was chosen. We preferred median to average to diminish the influence of other applications and OS interrupts as well as to improve reproducibility of timings between the runs of application. In particular, in the diagnostic boot of Windows, where the minimum of drivers and applications are loaded, we were getting the same number of cycles per iteration 70-80 out of 101 times. Timings in non-diagnostic boots had somewhat larger absolute values, however the relative performance remained unchanged and equally well reproducible.

4.1 Pattern Matching Overhead

The overhead associated with pattern matching may originate from several different sources:

- Naive (sequential and often duplicated) order of tests due to a pure library solution.
- Compiler's inability to inline the test expressed by the pattern in the left-hand side of the case clause (e.g. due to lack of [type] information or complexity of the expression).
- Compiler's inability to elide construction of pattern trees when used in the right-hand side of the case clause.

To estimate the overhead introduced by commonly used *patterns as objects* approach and our *patterns as expression templates* approach (§3.1), we implemented several simple functions with and without pattern matching. The handcrafted code we compared against was hand-optimized by us to render the same results, without changes to the underlying algorithm. Some functions were implemented in several ways with different patterns in order to show the impact of different patterns and pattern combinations on performance. The overhead of both approaches on a range of recent C++ compilers is shown in Figure 1.

		Patterns as Expr. Templates					Patterns as Objects				
		G++			Visual C++		G++			Visual C++	
Test	Patterns	4.5.2	4.6.1	4.7.2	10.0	11.0	4.5.2	4.6.1	4.7.2	10.0	11.0
factorial_0^*	1,v,_	**15%**	**13%**	**17%**	85%	35%	347%	408%	419%	2121%	1788%
factorial_1	1,v	0%	6%	0%	83%	21%	410%	519%	504%	2380%	1812%
factorial_2	1,n+k	7%	9%	6%	78%	%	797%	911%	803%	3554%	3057%
fibonacci	1,n+k	17%	**2%**	2%	62%	15%	340%	431%	395%	2730%	2597%
gcd_1	v,n+k,+	21%	25%	25%	309%	179%	1503%	1333%	1208%	8876%	7810%
gcd_2	1,n+k,_	5%	13%	19%	373%	303%	962%	1080%	779%	5332%	4674%
gcd_3	1,v	**1%**	0%	**1%**	38%	15%	119%	102%	108%	1575%	1319%
lambdas*	&,v,C,+	58%	54%	56%	**29%**	**34%**	837%	780%	875%	259%	289%
power	1,n+k	10%	8%	13%	50%	6%	291%	337%	338%	1950%	1648%

Figure 1. Pattern Matching Overhead

The experiments marked with ∗ correspond to the functions in §2. The rest of the functions, including all the implementations with "patterns as objects" approach, are available on the project's web page. The patterns involved in each experiment are abbreviated as following: **1** – value pattern; **v** – variable pattern; _ – wildcard pattern; **n+k** – n+k (application) pattern; **+** – equivalence combinator; **&** – address combinator; **C** – constructor pattern.

The overhead incurred by compile-time composition of patterns in *patterns as expression templates* approach is significantly smaller than the overhead of run-time composition of patterns in *patterns as objects* approach. In cases, shown in the table in bold, the compiler was able to eliminate the overhead entirely. In case of the "lambdas" experiment, the advantage was mainly due to the underlying type switch, while in the rest of the cases the generated code better utilized the instruction pipeline and the branch predictor.

In each experiment, the handcrafted baseline implementation was the same in both cases (compile-time and run-time composition) and reflected our idea of the fastest code without pattern matching describing the same algorithm. For example, gcd_3 was implementing the fast Euclidian algorithm with remainders, while gcd_1 and gcd_2 were implementing its slower version with subtractions. The baseline code was correspondingly implementing fast Euclidian algorithm for gcd_3 and slow for gcd_1 and gcd_2.

The comparison of the overhead incurred by both approaches would be incomplete without detailing the implementation of patterns as objects solution. In particular, dealing with objects in object-oriented languages often involves heap allocation, subtype tests, garbage collection etc., which all can significantly affect the performance. To make this comparison applicable to a wider range of object-oriented languages, we took the following precautions in the "objects as patterns" implementations:

- All the objects involved were stack-allocated or statically allocated. This measure was taken to avoid allocating objects on the heap, which is known to be much slower and is an optimization compilers of many object-oriented languages undertake.
- Objects representing constant values as well as patterns whose state does not change during pattern matching (e.g. wildcard and value patterns) were all statically allocated.
- Patterns that modify their own state were constructed only when they were actually used, since a successful match by a previous pattern may return early from the function.
- Only the arguments on which pattern matching was effectively performed were boxed into the object class hierarchy, e.g. in case of power function only the second argument was boxed.
- Boxed arguments were statically typed with their most derived type to avoid unnecessary type checks and conversions: e.g. object_of⟨**int**⟩&, which is a class derived from object to represent a boxed integer, instead of just object&.
- No objects were returned as a result of a function as in truly object-oriented approach that might require heap allocation.
- n+k patterns that effectively require evaluating a result of an expression where implemented with an additional virtual function instead that simply checks whether a result is a given value. This does not allow expressing all n+k patterns of *Mach7*, but was sufficient to express all those involved in the experiments and allowed us to avoid heap allocating the results.
- When run-time type checks were unavoidable (e.g. inside pattern::match implementation) we were comparing type-ids first, and only when the comparison failed were invoking the much slower dynamic cast to optimize the common case.

With these precautions in place, the main overhead of the patterns-as-objects solution was in the cost of a virtual function call (pattern::match) and the cost of run-time type identification and conversion on its argument – the subject. Both are specific to the approach and not to our implementation, so similar overhead is present in other object-oriented languages following this strategy.

4.2 Compilation Time Overhead

Several people expressed concerns about possible significant increase in the compilation time due to openness of our pattern matching solution. While this might be the case for some patterns that require a lot of compile time computations, it is not the case with any of the common patterns we implemented. Our patterns are simple top-down instantiations that rarely go beyond standard overload resolution or occasional enable_if condition. Furthermore, we compared the compilation time for each of the examples discussed in §4.1 with a handcrafted version.

		Patterns as Expr. Templates			Patterns as Objects		
		G++	Visual C++		G++	Visual C++	
Test	Patterns	4.7.2	10.0	11.0	4.7.2	10.0	11.0
$factorial_0^*$	1,v,_	1.65%	1.65%	2.95%	7.10%	**10.00%**	10.68%
$factorial_1$	1,v	2.46%	1.60%	10.92%	7.14%	0.00%	1.37%
$factorial_2$	1,n+k	2.87%	3.15%	3.01%	8.93%	4.05%	**3.83%**
fibonacci*	1,n+k	3.66%	1.60%	2.95%	11.31%	**4.03%**	1.37%
gcd_1^*	v,n+k,+	4.07%	4.68%	**0.91%**	9.94%	2.05%	8.05%
gcd_2	1,n+k,_	1.21%	1.53%	**0.92%**	8.19%	**2.05%**	**2.58%**
gcd_3	1,v	2.03%	3.15%	7.86%	5.29%	2.05%	0.08%
lambdas*	&,v,C,+	18.91%	7.25%	**4.27%**	4.57%	**3.82%**	0.00%
power	1,n+k	2.00%	6.40%	3.92%	8.14%	0.13%	4.02%

Table 1. Compilation Time Overhead

As can be seen in Table 1, the difference in compilation times was not significant – on average 3.99% for open patterns and 4.84% for patterns as objects, with patterns compiling faster in few cases indicated with bold font. The difference on real-world projects with significant percentage of non-pattern matching code will be less.

4.3 Multi-argument Hashing

To check the efficiency of hashing in the multi-argument *Match* statement (§3.5) we used the same class hierarchy benchmark we used to test the efficiency of hashing in type switch [40, §4.4]. The benchmark consists of 13 libraries describing 15246 classes totally. Not all the class hierarchies originated from C++, but all were written by humans and represent actual relationships in their problem domains.

While *Match* statement works with both polymorphic and non-polymorphic arguments, only the polymorphic arguments are taken into consideration for efficient type switching and thus efficient hashing. It also generally makes sense to apply type switching to non-leaf nodes of the class hierarchy only. 71% of the classes in the entire benchmarks suite were leaf classes. For each of the remaining 4369 non-leaf classes we created 4 functions, performing case analysis on derived class on a combination of 1, 2, 3 and 4 arguments respectively. Each of the functions was executed with different combinations of possible derived types, including, in case of repeated multiple inheritance, different sub-objects within the same type. There was 63963 different subobjects when the class hierarchies used repeated multiple inheritance and 38856 different subobjects when virtual multiple inheritance was used.

As with type switching, for each of the 4369 functions (per same number of arguments) we were measuring the number of conflicts m in cache – the number of entries mapped to the same location in cache by the optimal hash function. We then computed the percentage of functions that achieved a given number of conflicts, which we show in Figure 2.

We grouped the results in ranges of exponentially increasing size because we noticed that the number of conflicts per *Match* statement for multiple arguments was not as tightly distributed around 0 as it was for a single argument. The main observation however still holds: in most of the cases, we could achieve hashing without conflicts, as can be seen in the first column (marked [0]). The numbers are slightly better when virtual inheritance is used because the overall number of possible subobjects is smaller.

	N/m	[0]	[1]	$\cdots$ 10]	$\cdots$ 100]	$\cdots$ 1000]	$\cdots$ 10000]	>10000
Repeated	1	88.37%	10.78%	0.85%	0.00%	0.00%	0.00%	0.00%
	2	76.42%	5.51%	10.60%	4.89%	2.22%	0.37%	0.00%
	3	65.18%	0.00%	15.04%	8.92%	5.83%	5.03%	0.00%
	4	64.95%	0.00%	0.14%	14.81%	7.57%	12.54%	0.00%
Virtual	1	89.72%	9.04%	1.24%	0.00%	0.00%	0.00%	0.00%
	2	80.55%	4.20%	8.46%	4.59%	1.67%	0.53%	0.00%
	3	71.26%	0.37%	12.03%	7.32%	4.87%	4.16%	0.00%
	4	71.55%	0.00%	0.23%	11.83%	6.49%	9.90%	0.00%

Figure 2. Percentage of N-argument *Match* statements with given number of conflicts (m) in cache

4.4 Comparison to Multiple Dispatch Alternatives

Type switching on multiple arguments can be seen as a form of multiple dispatch. Here, we compare the efficiency of type switching on multiple arguments in comparison to alternatives based on double, triple and quadruple dispatch [16], as well as our own implementation of open multi-methods for C++ [36].

The need for multiple dispatch rarely happens in practice, diminishing with the number of arguments involved in dispatch. Muschevici et al [27] studied a large corpus of applications in 6 languages and estimate that single dispatch amounts to about 30% of all the functions, while multiple dispatch is only used in 3% of functions. In application to type switching, this indicates that we can expect case analysis on dynamic type of a single argument much more often than that on dynamic types of two or more arguments. Note, however, that this does not mean that pattern matching in general reflects the same trend as additional arguments are often introduced into the *Match* statement to check some relational properties. These additional arguments are typically non-polymorphic and thus do not participate in type switching, which is why in this experiment we only deal with polymorphic arguments.

Figure 3 contains 4 bar groups that corresponding to the number of arguments used for multiple dispatch. Each group contains 3 wide bars representing the number of cycles per iteration it took N-Dispatch, Open Type Switch and Open Multi-Methods solution to perform the same task. Each of the 3 wide bars is subsequently split into 5 narrow sub-bars representing performance achieved by G++ 4.5.2, 4.6.1, 4.7.2 and Visual C++ 10 and 11 in that order from left to right.

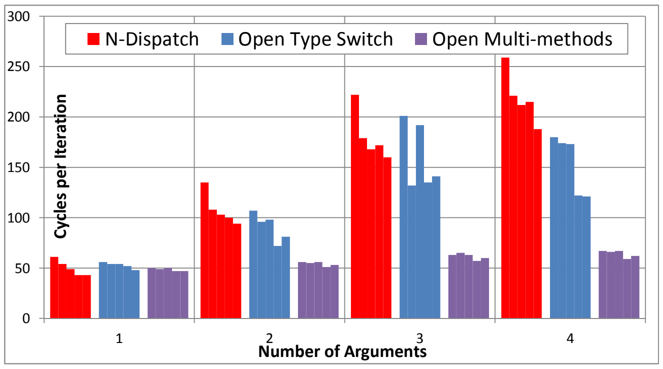

Figure 3. N-argument *Match* statement vs. visitor design pattern and open multi-methods

Open multi-methods provide the fastest performance because the dispatch is implemented with an N-dimensional array lookup, requiring only $4N + 1$ memory references before an indirect call. N-Dispatch provides the slowest solution, requiring $2N$ virtual function calls (accept/visit per each dimension). Open type switch falls in between the two, thanks to its efficient hashing combined with a jump table.

In terms of memory, given a class hierarchy of n classes (actually n subobjects in the subobject graph) and a multiple dispatch on N arguments from it, all 3 solutions will require memory proportional to $O\left(n^N\right)$. More specifically, if δ is the number of bytes used by a pointer, then each of the approaches will use:

- Open Multi-Methods: $\delta\left(n^N + Nn + N\right)$
- N-Dispatch: $\delta\left(n^N + n^{N-1} + \cdots + n^2 + n\right)$
- Open Type Switch: $\delta\left((2N+3)\,n^N + N + 7\right)$

bytes of memory. In all 3 cases the memory counted represents the non-reusable memory specific to the implementation of a single function dispatched through N polymorphic arguments. Note that n is a variable here since new classes may be loaded at run-time through dynamic linking in all 3 solutions, while N is a constant, representing the number of arguments to dispatch on.

The memory used by each approach is allocated at different stages. The memory used by the virtual tables involved in the N-dispatch solution as well as dispatch tables used by open multi-methods will be allocated at compile/link time and will be reflected in the size of the final executable. Open multi-methods might require additional allocations and/or recomputation at load time to account for dynamic linking. In both cases, the memory allocated covers all possible combinations of n classes in N argument positions. In case of open type switch, the memory is only allocated at run-time and grows proportionally to the number of actual argument combinations seen by the type switch (§3.5.1). Only in the worst case when all possible combinations have been seen by the type switch does it reach the size described by the above formula. This is an important distinction as in many applications all-possible combinations will never be seen: for example, in a compiler the entities representing expressions and types might all be derived from a common base class, however they will rarely appear in the same type switch together.

There is also a significant difference in the ease of use of these solutions. N-Dispatch is the most restrictive solution as it is intrusive (and thus cannot be applied retroactively), hinders extensibility (by limiting the set of distinguishable cases) and is surprisingly hard to teach students. While analyzing Java idioms used to emulate multiple dispatch in practice, Muschevici et al [27, Figure 13] noted that there are significantly more uses of cascading instanceof in the real code than the uses of double dispatch, which they also attribute to the obscurity of the second idiom. Both N-Dispatch and open multi-methods also introduce control inversion in which the case analysis is effectively structured in the form of callbacks. Open multi-methods are also subject to ambiguities, which have to be resolved at compile time and in some cases might require the addition of numerous overriders. Neither is the case with open type switch where the case analysis is performed directly, while ambiguities are avoided by the use of first-fit semantics.

4.5 Rewriting Haskell Code in C++

For this experiment we took an existing code in Haskell and asked its author to rewrite it in C++ with *Mach7*. The code in question is a simple peephole optimizer for an experimental GPU language called *Versity*. We assisted the author along the way to see which patterns he uses and what kind of mistakes he makes.

Somewhat surprising to us we found out that pattern-matching clauses generally became shorter, but their right-hand side became longer. The shortening of case clauses was perhaps specific to this application and mainly stemmed from the fact that Haskell does not support equivalence patterns or equivalence combinator and had to use guards to relate different arguments. This was particularly cumbersome when optimizer was looking at several arguments of several instructions in the stream, e.g.:

```
peep2 (x1:x2:xs) =
  case x1 of
    InstMove a b →
      case x2 of
        InstMove c d | (a ==d) && (b ==c) → peep2 $ x1:xs
```

compared to *Mach7* version:

```
Match(∗x1,∗x2) {
  Case(C⟨InstMove⟩(a,b), C⟨InstMove⟩(+b,+a)) ...
```

Haskell also requires to use wildcard pattern in every unused position of a constructor pattern (e.g. InstBin _ _ _ _), while *Mach7* allows one to omit all the trailing wildcards in constructor patterns (e.g. C⟨InstBin⟩()). The use of named patterns avoided many repeated pattern expressions and actually improved both performance and readability:

```
auto either  = val(src ) || val (dst);
Match(inst) {
  Case(C⟨InstMove⟩(_,        either )) ...
  Case(C⟨InstUn⟩  (_, _,     either )) ...
  Case(C⟨InstBin⟩ (_, _, _,  either )) ...
} EndMatch
```

The disadvantage for *Mach7* was coming after pattern matching, as we had to explicitly manage memory when inserting, removing or replacing instructions in the stream as well as explicitly manage the stream itself. Eventually we could hide some of this boilerplate behind smart pointers and other standard library classes.

During the initial rewrite into C++ the developer simply mapped Haskell patterns into their *Mach7* equivalents, at which point we intervened and showed how some of the patterns can be expressed simpler. We reiterated the process until we could not improve the patterns ourselves without getting into the details of the actual simplifier.

We have yet to make a performance comparison, but based on the performance of the open type switch [40] and the small overhead of our patterns, we expect the performance to be comparable.

4.6 Limitations

While our patterns can be saved in variables and passed over to functions, they are not true first-class citizens. In fact, we believe that the notion of being a first-class citizen in a language should be updated to take parametric polymorphism of C++ into account. The reason is that as is, our solution does not allow for composing patterns based on user's input (e.g. by letting the user type in a pattern to match against). This can potentially be solved by mixing "patterns as objects" approach in, however the performance overhead we saw in §4.1 is too costly to be adopted.

5. Related Work

Language support for pattern matching was first introduced for string manipulation in COMIT[48], which subsequently inspired similar primitives in SNOBOL[10]. SNOBOL4 had string patterns as first-class data types providing operations of concatenation and alternation. The first reference to modern pattern-matching constructs as seen in functional languages is usually attributed to Burstall's work on structural induction[4]. Pattern matching was further developed by the functional programming community, most notably ML[12] and Haskell[15]. In the context of object-oriented programming, pattern matching has been first explored in Pizza[31] and Scala[9, 30]. The idea of first-class patterns dates back at least to Tullsen's proposal to add them to Haskell [43]. The calculus of such patterns has been studied in detail by Jay [19, 20].

There are two main approaches to compiling pattern-matching code: the first is based on *backtracking automata* and was introduced by Augustsson[1], the second is based on *decision trees* and was first described by Cardelli[5]. The backtracking approach usually generates smaller code [21], whereas the decision tree approach produces faster code by ensuring that each primitive test is only performed once [25].

There have been several attempts to bring pattern matching into various languages in a form of a library. They differ on which abstractions of the host language were used to encode the patterns and the match statement. *MatchO* was one of the first such attempts for Java [46]. The approach follows the *patterns as objects* strategy. *Functional C#* was a similar approach, bringing pattern matching to C# as a library[34]. The approach uses lambda expressions and chaining of method calls to create a structure that is then evaluated at run time for the first successful match. In the functional community, Rhiger explored introduction of pattern matching into Haskell as a library [38]. He uses functions to encode patterns and pattern combinators, which allows him to detect pattern misapplication errors at compile time through the Haskell type system. *Racket* has a powerful macro system that allows it to express open pattern matching in the language entirely as a library [42]. The solution is remarkable in that unlike most of the library approaches to open pattern matching, it does not rely on naïve backtracking and, in fact, encodes the optimized algorithm based on backtracking automata [1, 21]. *Grace* is another programming language that provides a library solution to pattern matching through objects [14]. Similar to other control structures in the language, Grace encodes the match statement with partial functions and lambda expressions, while patterns are encoded as objects.

Multiple language extensions have been developed to provide pattern matching into a host language in a form of a compiler, preprocessor or tool. *Prop* brought pattern matching and term rewriting into C++ [22]. It did not offer first-class patterns, but supported most of the functional-style patterns and provided an optimizing compiler for both pattern matching and garbage-collected term rewriting. *App* was another pattern-matching extension to C++ [29] that mainly concentrated on providing syntax for defining and pattern matching on algebraic data types. *Tom* is a pattern-matching compiler that brings a common pattern-matching and term-rewriting syntax into Java, C and Eiffel. Thanks to its distinct syntax, it is transparent to the semantics of the host language and can be implemented as a preprocessor to many other languages. Tom neither supports first-class patterns, nor is open to new patterns. *Matchete* is a language extension to Java that brings together different flavors of pattern matching: functional-style patterns, Perl-style regular expressions, XPath expressions, Erlang's bit-level patterns etc. [13]. The extension does not try to make patterns first-class citizens, but instead concentrates on implementing existing best practices and their tight integration into Java. *OOMatch* is another Java extension that brings pattern matching and multiple dispatch close together [39]. The approach generalizes multiple dispatch by offering to use patterns as multi-method arguments and then orders overriders based on the specificity of their arguments. Similar to others, the approach only deals with a limited set of built-in patterns.

Thorn is dynamically typed scripting language that provides first-class patterns [2]. The language defines a handful of atomic patterns and pattern combinators to compose them, and, similar to Newspeak and Grace, uses the duality between partial functions and patterns to support user-defined patterns.

When a class hierarchy is fixed, one can design a pattern language that involves semantic notions represented by the hierarchy. Pirkelbauer devised a pattern language for Pivot [8] capable of representing various entities in a C++ program using syntax very close to C++ itself. The patterns were translated with a tool into a set of visitors implementing the underlying pattern-matching semantics efficiently [35].

6. 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 an example. It is non-intrusive and can be applied retroactively. The library provides efficient and expressive matching on multiple subjects 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. *Mach7* pattern matching code performs reasonably compared to open multi-methods and visitors, demonstrating the effectiveness of the library-based approach.

The work presented here continues our research on pattern matching for C++ [40]. 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 would like to experiment with other kinds of patterns, including those defined by the user; look at the interaction of patterns with other facilities in the language and the standard library; make views less ad hoc etc.

References

[1] L. Augustsson. Compiling pattern matching. In *Proc. of a conference on Functional programming languages and computer architecture*, pp 368–381, New York, USA, 1985. Springer-Verlag Inc.

[2] B. Bloom and M. J. Hirzel. Robust scripting via patterns. In *Proc. ACM DLS'12*, pp 29–40, NY, USA.

[3] K. Bruce, L. Cardelli, G. Castagna, G. T. Leavens, and B. Pierce. On binary methods. *Theor. Pract. Object Syst.*, 1(3):221–242, 1995.

[4] R. M. Burstall. Proving properties of programs by structural induction. *Computer Journal*, 1969.

[5] L. Cardelli. Compiling a functional language. In *Proc. ACM LFP'84*, pp 208–217.

[6] P. Cuoq, J. Signoles, P. Baudin, R. Bonichon, G. Canet, L. Correnson, B. Monate, V. Prevosto, and A. Puccetti. Experience report: Ocaml for an industrial-strength static analysis framework. In *Proc. ACM ICFP'09*, pp 281–286, New York, USA.

[7] S. Don, G. Neverov, and J. Margetson. Extensible pattern matching via a lightweight language extension. In *Proc. ACM ICFP'07*, pp 29–40.

[8] G. Dos Reis and B. Stroustrup. A principled, complete, and efficient representation of C++. In *Joint ASCM'09 and MACIS'09*, pp 407–421.

[9] B. Emir. *Object-oriented pattern matching*. PhD thesis, Lausanne, 2007.

[10] D. J. Farber, R. E. Griswold, and I. P. Polonsky. Snobol , a string manipulation language. *J. ACM*, 11:21–30, January 1964.

[11] F. Geller, R. Hirschfeld, and G. Bracha. *Pattern Matching for an object-oriented and dynamically typed programming language*. Technische Berichte, Universität Potsdam. Univ.-Verlag, 2010.

[12] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. In *Proc. ACM POPL'78*, pp 119–130, New York, USA.

[13] M. Hirzel, N. Nystrom, B. Bloom, and J. Vitek. Matchete: Paths through the pattern matching jungle. In *Proc. PADL'08*, pp 150–166.

[14] M. Homer, J. Noble, K. B. Bruce, A. P. Black, and D. J. Pearce. Patterns as objects in Grace. In *Proc. ACM DLS'12*, pp 17–28.

[15] P. Hudak, H. Committee, P. Wadler, and S. Jones. *Report on the Programming Language Haskell: A Non-strict, Purely Functional Language : Version 1.0*. ML Library. Haskell Committee, 1990.

[16] D. H. H. Ingalls. A simple technique for handling multiple polymorphism. In *Proc. ACM OOPSLA'86*, pp 347–349, New York, USA.

[17] International Organization for Standardization. *ISO/IEC 14882:2011: Programming languages: C++*. Geneva, Switzerland, 2011.

[18] J. Järvi, J. Willcock, H. Hinnant, and A. Lumsdaine. Function overloading based on arbitrary properties of types. *C/C++ Users Journal*, 21(6):25–32, June 2003.

[19] B. Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[20] B. Jay and D. Kesner. First-class patterns. *J. Funct. Program.*, 19(2):191–225, Mar. 2009.

[21] F. Le Fessant and L. Maranget. Optimizing pattern matching. In *Proc. ACM ICFP'01*, pp 26–37, New York, USA.

[22] A. Leung. Prop: A C++ based pattern matching language. Technical report, Courant Institute, New York University, 1996.

[23] M. G. M. A computer-oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd, Ottawa, Canada, 1966.

[24] L. Maranget. Compiling lazy pattern matching. In *Proc. ACM LFP'92*, pp 21–31, New York, USA.

[25] L. Maranget. Compiling pattern matching to good decision trees. In *Proc. ACM ML'08*, pp 35–46, New York, USA.

[26] Y. Minsky. Caml trading - experiences with functional programming on Wall Street. *JFP*, 18:553–564, 2008.

[27] R. Muschevici, A. Potanin, E. Tempero, and J. Noble. Multiple dispatch in practice. In *Proc. ACM OOPSLA'08*, pp 563–582.

[28] R. Nanavati. Experience report: a pure shirt fits. In *Proc. ACM ICFP'08*, pp 347–352, New York, USA.

[29] G. Nelan. An algebraic typing & pattern matching preprocessor for C++, 2000. http://www.primenet.com/ georgen/app.html.

[30] M. Odersky, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. Mcdirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. An overview of the Scala programming language (2nd edition). Technical report, EPTF, 2006.

[31] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *In Proc. ACM POPL'97*, pp 146–159.

[32] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[33] N. Oosterhof. Application patterns in functional languages, 2005.

[34] E. Pentangelo. Functional C#. `http://functionalcsharp.codeplex.com/`, 2011.

[35] P. Pirkelbauer. *Programming Language Evolution and Source Code Rejuvenation*. PhD thesis, Texas A&M University, December 2010.

[36] P. Pirkelbauer, Y. Solodkyy, and B. Stroustrup. Open multi-methods for C++. In *Proc. ACM GPCE'07*, pp 123–134, New York, USA.

[37] L. Puel and A. Suarez. Compiling pattern matching by term decomposition. *J. Symb. Comput.*, 15(1):1–26, Jan. 1993.

[38] M. Rhiger. Type-safe pattern combinators. *J. Funct. Program.*, 19(2):145–156, Mar. 2009.

[39] A. Richard. OOMatch: pattern matching as dispatch in Java. Master's thesis, University of Waterloo, October 2007.

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

[41] A. Sutton, B. Stroustrup, and G. Dos Reis. Concepts lite: Constraining templates with predicates. Technical Report WG21/N3580, JTC1/SC22/WG21 C++ Standards Committee, 2013.

[42] S. Tobin-Hochstadt. Extensible pattern matching in an extensible language. September 2010.

[43] M. Tullsen. First class patterns. In *Proc. PADL'00*, pp 1–15.

[44] D. Vandevoorde and N. Josuttis. *C++ templates: the complete guide*. Addison-Wesley, 2003.

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

[46] J. Visser. Matching objects without language extension. *Journal of Object Technology*, 5.

[47] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proc. ACM POPL'87*, pp 307–313, New York, USA.

[48] V. H. Yngve. A programming language for mechanical translation. *Mechanical Translation*, 5:25–41, July 1958.