Hi, my name is Yuriy and today I would like to present the work we did in collaboration with Dr. Dos Reis and Dr. Stroustrup, which continues our research on pattern matching for C++.
Our Mach7 library provides an open solution to pattern matching in standard C++, by making all the patterns in the library user-definable. The patterns are first-class citizens, which can be saved in variables and passed to functions. The solution is also type safe where incorrect pattern applications are reported at compile time. The solution is non-intrusive and can be applied retroactively. It builds on top of our efficient type switch construct and provides a pattern-matching solution that is faster than existing alternatives to open pattern matching in C++. 
Introduction

- **What is a pattern?**
  - a term representing an immediate predicate on an implicit argument
- **What is pattern matching?**
  - a way of checking the structure of data and decomposing it into subcomponents
- **Examples of patterns**
  - Wildcards, Variables, Values, Regular Expressions, Terms of the above etc.

So what is a pattern? Languages with the closed set of patterns usually define them inductively by enumerating the list of primitive patterns and means of their composition. In a fully open setting by pattern we mean any term representing an immediate predicate on an implicit argument. Implicitness of the argument is what sets apart pattern from lambda-predicates of C++11.

Similarly to other languages, by pattern matching we mean a way of checking the structure of run-time values and decomposing them into subcomponents. The typical examples of patterns are wildcards, variables, regular expressions, tree patterns etc.
To give examples of patterns and pattern matching, imagine that you would like to implement an interpreter (or calculator) of a simple expression language represented by the following grammar. In functional languages, a term in such language will typically be represented with a recursive Algebraic Data Type (ADT) with variants corresponding to production rules. Functional languages then implement various operations on expression terms (like evaluation, printing or transformation) by simple case analysis on the variants with variables used to bind to corresponding subcomponents. An evaluation function, for example, is a simple recursion that follows the structure of the term.
Variables are not the only entities that can be bound to subcomponents and other patterns can be used instead to check the structure of those subcomponents. This leads to nesting of patterns as demonstrated here with function `collect` that extracts common factor from a sum. This is also where implicitness of the argument makes the solution less verbose than same solution expressed with lambda functions.

```
let collect e =
    match e with
    | Plus(Times(x1,a), Times(x2,b)) when x1=x2
        -> Times(x1,Plus(a,b))
    | Plus(Times(a,x1), Times(b,x2)) when x1=x2
        -> Times(Plus(a,b),x1)
    | e -> e
;;
```
The nesting of patterns can also be used to establish relations between multiple values. In a canonical example all the arguments are explicitly placed into a tuple, which is then decomposed with tuples of patterns.

```plaintext
let rec equal e1 e2 =
  match (e1,e2) with
  | (Value(v1), Value(v2)) -> v1 == v2
  | (Plus (a1, b1), Plus (a2, b2))
  | (Minus (a1, b1), Minus (a2, b2))
  | (Times (a1, b1), Times (a2, b2))
  | (Divide(a1, b1), Divide(a2, b2))
    -> (equal a1 a2) && (equal b1 b2)
  | _ -> false
```

Object-oriented languages, where variants are often represented with derived classes, usually don’t provide such expressive facilities. The case analysis is usually performed either with dedicated virtual functions or a reusable visitor design pattern. Both are intrusive, introduce control inversion and potentially inhibit local reasoning. Visitors are also notoriously hard to teach to novices. In the absence of dedicated predicates, nested matching is usually explicit via nested if statements, which is verbose. Relational matching is often achieved with double dispatch, which bears all the problem of the visitor design pattern it is based on.
So why bring pattern matching into an object-oriented language? Because the code for many applications is so notoriously shorter and easier to understand and maintain that people often opt for functional languages for certain kind of applications just because of pattern matching. The example here demonstrates a complete implementation of insertion into a red-black tree in a functional language and an incomplete implementation of the same functionality in an imperative language. Which of these would you expect to have more bugs?
An ultimate language feature of bringing pattern matching into C++ would have to satisfy certain design criteria and ideally meet some goals. Following the ideals of D&E book for C++, the feature would have to support the existing C++ object model, i.e. be able to deal with multiple inheritance, work in the presence of dynamic linking, be applicable to both user-defined and built-in types etc. To be adopted, it will also have to be comparable or faster than any known workarounds.

Its design ideals include intuitiveness, simplicity and ease of teaching present in the pattern matching facilities of functional languages. It should also be a direct show of intent and checkable (if that can be done without sacrificing the openness). The important ideal we would like to achieve in the light of what we could do with Mach7 library is to maintain openness of both set of patterns as well as set of classes analyzed.
Coming back to our expression language, here is almost everything the user has to write to implement a fully functional evaluator of expression terms in Mach7 side compared with the original OCaml code. It is also as efficient, while does not assume a closed world. The only missing bit is binding definitions, which tell the library how to decompose an object of a given type. These definitions are made once for the entire hierarchy, can be parameterized and are made retroactively. They are non-intrusive and fully respect encapsulation rules of C++.

The structural decomposition in Mach7 is achieved with the help of C<> template representing a constructor pattern. It can be composed of other patterns, like the explicit variable patterns here, and knows how to use the bindings to access corresponding subcomponents.

Similarly to OCaml solution, our solution is:
- general
- concise and direct
- avoids control inversion
- open to class extensions
- intuitive and thus easier to teach
The nesting of patterns in Mach7 is further demonstrated in our implementation of the collect function. Remember that unlike solution in OCaml, this is a library solution where all the patterns are user-definable. It has the negative implication that the variable patterns have to be explicitly introduced, however it also allows us to create custom patterns and pattern combinators which can further shorten the code. This is the case in this example, where we use a user-defined equivalence combinator + that turns binding uses of a variable pattern into non-binding ones.
Mach7’s approach to relational matching is slightly unorthodox as instead of wrapping multiple arguments into tuples we provide a Match statement capable of taking multiple arguments. This is done intentionally to allow patterns that might query the dynamic type of the subject to benefit from the underlaid type switch. Unlike OCaml’s solution, our solution is also fully open as it allows for both new patterns and pattern combinators as well as is open to new classes, including those linked dynamically. Note also how the use of equivalence combinator + actually shortens the code in comparison to OCaml’s code with guards.
The equivalence combinator is only one of the pattern combinators commonly used in other languages (e.g. Thorn, Grace). Pattern combinators constitute operators for creating new patterns and modifying the meaning of existing ones. A guard combinator, for example, combines a pattern with a lazily evaluated guard into a guard pattern. Similarly, conjunction, disjunction and negation combinators allow one to compose patterns logically. Mach7 also introduces a few combinators specific to C++, which help deal with pointers.
Combined together, you can see that our open patterns allow us to express the functional solution to balancing red-black trees as tersely as it is known in the functional community.
Rod Burstall coined the idea of constructor patterns to explicitly facilitate structural induction on terms in order to be able to prove some properties about programs. With Peano construction of natural numbers, the idea was soon generalized to n+k patterns in order to facilitate the mathematical induction on natural numbers. Further attempts to generalize this idea included application patterns, which try to give such patterns equational semantics. The semantics of application patterns is obvious when the solution to corresponding equation is unique or absent, but when multiple solution exist, additional, often obscure and complicated rules, have to be put in place. Instead of trying to generalize algebraic decomposition along the equational route, we suggest to interpret them as notational patterns that help the programmer decompose parts of a mathematical objects with a well established notation known for such entity. For example, n/m is a common notation for rational numbers, which can be used to decompose an object representing a rational number into nominator and denominator. Similarly, a+bi is a common notation for decomposing complex numbers, while 3q+r can be used to obtain quotient and remainder of dividing a number by 3. More complex mathematical objects may have even more complex notations, like for example, 2D line that can be represented with slope-intercept, linear equation or two points form. Note that in the last 2 cases the equality sign is already embedded into the notation, making it hard to consider the equational semantics.
Interestingly, the notational semantics of generalized n+k patterns allows us to express the useful cases of application patterns with their equational semantics. The following function implements fast algorithm for computing \( x^n \) and mimics almost one to one the mathematical definition from the book. The explicitness of variable pattern helps in this case as we can take the type of the underlying variable into account when providing the semantics of matching against a given expression. This allows us to reject matches for integral types and simply solve the equation when underlying type is a field. This is further demonstrated in the slow subtraction algorithm for GCD computation where for unsigned \( x \) the expression \( b+x \) will match the first argument \( a \) only when \( b > a \).

```cpp
double power(double x, int n)
{
    var<int> m;
    Match(n)
    {
        Case(0) return 1.0;
        Case(1) return x;
        Case(2*m) return sqr(power(x,m));
        Case(2*m+1) return x*sqr(power(x,m));
    }
    EndMatch
}

size_t gcd(const size_t a, const size_t b)
{
    var<size_t> x;
    Match(a,b)
    {
        Case(_,a) return a;
        Case(_,a+x) return gcd(a,x);
        Case(b+x,_) return gcd(b,x);
    }
    EndMatch
}
```

- **Equational reasoning**
  - Since \( n \) is of integral type:
    \[
    x^n = \begin{cases} 
      1 & n = 0 \\
      x & n = 1 \\
      x^m \cdot x^m & 2m = n/2 \\
      x^m \cdot x^m & 2m + 1 = n \\
    \end{cases}
    \]
  - \( n = 2^m + 1 \) \( \Rightarrow \) \( n \) must be odd and \( m = (n-1)/2 \)

- **Type matters**
  - \( b+x=a \) for unsigned \( x \) will match only when \( b > a \)
As we mentioned, the library does not rely on a set of pre-existing patterns and can be used with new patterns that are easy to create. Using the new regular expression class of C++11 we could quickly create a regular expression pattern that can be composed with other patterns and used inside the Match statement. The patterns can also be saved into variables as demonstrated here with month and day and then reused inside other patterns. The Match statement you see can be used to parse and check the string for being a local phone, a toll-free phone and a valid date.

```cpp
var<int> n, m, y, d, m;
auto month = m | m > 0 && m < 13; // Save pattern to variable
auto day = d | d > 0 && d < 31; // Day pattern
Match(s)
{
            any({800, 888, 877, 866, 855}), n, m)) // Toll-free phone
    Case(rex("([0-9]{4})-([0-9]{2})-([0-9]{2})",
            y, month, d | d > 0 && d <= 31)) // Date
    Otherwise() // Something else
}
EndMatch
```

- All Patterns in the library are user-defined
- Patterns can be saved in variables and passed to functions
The idea of open patterns is not new and has been explored before. In object-oriented languages it is usually based on two interfaces: object and pattern. The object interface is used to check equivalence of two values, while the pattern interface provides match function that is implemented by each specific pattern kind. The patterns in this case are composed at run-time and allow for dynamic composition of patterns based on user input.

The problem with this solution is that it is intrusive as it assumes all the classes that participate in pattern matching to be derived from a single base class. Type errors are only discovered at run-time. Most importantly however, the approach is extremely slow in comparison to handcrafted solution, as we will see shortly.
Patterns as Expression Templates

```
template <typename... P>
struct some_pattern : std::tuple<P...> {
  static_assert(conjunction<is_pattern<P>...>::value,"Not a pattern");
  some_pattern(P&&... p);
  template <typename S> struct accepted_type_for { typedef S type; };
  template <typename S> bool operator()(const S& s) const;
};
```

Compile-time composition of patterns

Pros
- Open patterns
- First-class patterns
- Non-intrusive
- Type errors at compile-time

Cons
- Compile-time composition

Our patterns are also implemented as objects, however these objects do not have to belong to any class hierarchy. Instead they all have to model a Pattern concept, which typically amounts to implementing a constructor taking sub-patterns, a type-function to benefit from type switching and an application operator implementing the actual matching logic.

The solution is non-intrusive as neither values nor patterns have to belong to a given class hierarchy. Incorrect applications of patterns to subjects or invalid pattern compositions are reported at compile time (mostly with nice messages from static_assert, but occasionally with cryptic ones). The main disadvantage in comparison to the “Patterns as Objects” approach is that because pattern composition happens at compile time, the patterns cannot be composed based on user input. This is the case however with most of the languages with predefined set of patterns, and is generally expected from a pattern matching mechanism of a language.
To estimate the overhead of both approaches to pattern matching we compared a code written with patterns and an equivalent code without patterns implementing the same algorithm. As can be seen on the right, patterns implemented with “Patterns as objects” approach introduce significant overhead over the hand-crafted code. This is not the case with our approach, where the overhead is fairly small. In some cases, our approach even produced faster code, indicated in lighter blue font in the table.
Several people have also expressed their concerns that since our solution is based on templates, the compilation times will increase significantly. We measured the impact of compiling both approaches against compiling the hand-crafted code and did not find any significant increase in compilation times. This is expected as implementation of our patterns does not rely on complex meta-programming and in most of the cases amount to straightforward top-down instantiation of templates.
Finally, in application to relational matching we wanted to see how our solution compares to visitor-based implementations (double, triple, quadruple dispatch) as well as our own implementation of multi-methods for C++, which enable such relational checks.

It is easy to see that the solution based on the visitor design pattern was generally slower than the same solution based on pattern matching, which in turn was slower than the multi-methods based solution.

While all the solutions used the amount of memory of the same magnitude $O(n^N)$, the coefficients were quite different. Contrary to the common belief, multi-methods take less memory than the equivalent solution based on visitor design pattern. While type switch was using the most memory, the values are written for the worst case: the size of type switch grows proportionally to the number of actual argument pairs seen, while both N-Dispatch and open multi-methods essentially pre-allocate for the worst case all the time.
In conclusion, we presented a solution to open pattern matching in C++, which allows for both openness of patterns and objects being analyzed. It is faster than alternatives and can be applied retroactively. The current solution is a library implemented in standard C++ that is available as open source under BSD license.

The current solution is already competitive & we expect the forthcoming built-in support to be even better. In the meantime we plan to use it as a base line for features and performance evaluation of the actual language solution. We hope that experiences obtained in this implementation will also convince authors of future object-oriented languages to consider including pattern matching a-priory as many of the techniques used are applicable to broader set of languages.
We would like to acknowledge numerous individuals who helped with this publication, the library and evaluation of its use. I would also like to acknowledge my new employer, Microsoft, for accommodating this travel.