Technical Report

An Elegant and Efficient Pattern Matching Library for C++

accepting aint no visitors

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 greatly simplifies code. We present functional-programming-style pattern matching for C++ implemented as a library. The library provides a uniform notation for matching against open hierarchy of run-time polymorphic classes as well as closed set of classes (including classes tagged by user and discriminated unions) for which compile-time polymorphism can be used. The library integrates well with programming styles supported by C++, in particular it supports virtual and repeated multiple inheritance and can be used in generic code.

Our library equals or outperforms the visitor design pattern, as commonly used for pattern-matching scenarios in C++, and for many use cases it equals or outperforms equivalent code in languages with built-in pattern matching. Our solution better addresses more problems than the visitor design pattern does: it is non-intrusive and does not have extensibility restrictions. It also avoids control inversion and can be used in pattern-matching scenarios that visitors are ill suited for. Code using patterns is significantly more concise and easier to comprehend than alternative solutions in C++.

Implementing pattern matching as a library allows us to experiment with syntax, implementation algorithms, and use while preserving benefit from the performance and portability provided by industrial compilers and support tools. The solution approach can be reused in other object-oriented languages to implement *type switching*, *type testing*, *pattern matching* and *multiple dispatch* efficiently.

The library was motivated by and is used for applications involving large, typed, abstract syntax trees.

Categories and Subject Descriptors D [*3*]: 3

General Terms Languages, Design

Keywords Pattern Matching, Type Switching, Visitor Design Pattern, Expression Problem, Memoization, C++

1. Introduction

Pattern matching is an abstraction supported by many programming languages. It allows the user tersely to describe a (possibly infinite) set of values accepted by the pattern. A *pattern* represents a predicate on values, and is usually much more concise and readable than the equivalent predicate spelled out as imperative code.

Popularized by functional programming community, most notably Hope[5], ML[39], Miranda[57] and Haskell[28], for providing syntax very close to mathematical notations. From there, it has found its way into many imperative programming languages e.g. Pizza[41], Scala[42], Fortress[50], as well as dialects of Java[32, 35], C++[33], Eiffel[40] and others. It is relatively easy to provide a form of pattern matching when designing a new language, but to introduce it into a language in widespread use is a challenge. The obvious utility of the feature may be compromised by the need to fit into the language's syntax, semantics, and tool chains. A prototype implementation requires more effort than for an experimental language and is harder to get into use because mainstream users are unwilling to try non-portable, non-standard, and unoptimized features.

To balance the utility and effort we decided to take the Semantically Enhanced Library Language (SELL) approach[54]. We provide the general-purpose programming language with a library, extended with a tool support. This will typically (as in this case) not provide you 100% of the functionality that a language extension would do, but it allows experimentation and special-purpose use with existing compilers and tool chains. With pattern matching, we managed to avoid external tool support by relying on some pretty nasty macro hacking to provide a conventional and convenient interface to an efficient library implementation.

Our current solution is a proof of concept that sets a minimum threshold for performance, brevity, clarity and usefulness of a language solution for pattern matching in C++. It provides full functionality, so we can experiment with use of pattern matching in C++ and with language alternatives. To give an idea of what our library offers, consider an example from a domain where pattern matching is considered to provide terseness and clarity – compiler construction. Consider for example a simple language of expressions:

$$exp ::= val \mid exp + exp \mid exp - exp \mid exp * exp \mid exp/exp$$

An OCaml data type describing this grammar as well as a simple evaluator of expressions in it, can be declared as following:

```
type expr = Value of int
          | Plus  of expr * expr | Minus  of expr * expr
          | Times of expr * expr | Divide of expr * expr
          ;;

let rec eval e =
  match e with
          Value v       → v
        | Plus  (a, b) → (eval a) + (eval b)
        | Minus (a, b) → (eval a) − (eval b)
        | Times  (a, b) → (eval a) * (eval b)
        | Divide (a, b) → (eval a) / (eval b)
        ;;
```

The corresponding C++ data types would most likely be parameterized, but for now we will just use simple classes:

```
struct Expr { virtual ~Expr() {} };
struct Value  : Expr { int value; };
struct Plus   : Expr { Expr* exp1; Expr* exp2; };
struct Minus  : Expr { Expr* exp1; Expr* exp2; };
struct Times  : Expr { Expr* exp1; Expr* exp2; };
struct Divide : Expr { Expr* exp1; Expr* exp2; };
```

Using our library, we can express matching about as tersely as OCaml:

```
int eval(const Expr* e)
{
    Match(e)
    {
      Case(Value,  n)    return n;
      Case(Plus,   a, b) return eval(a) + eval(b);
      Case(Minus,  a, b) return eval(a) − eval(b);
      Case(Times,  a, b) return eval(a) * eval(b);
      Case(Divide, a, b) return eval(a) / eval(b);
    }
    EndMatch
}
```

To make the example fully functional we need to provide mappings of binding positions to corresponding class members:

```
template ⟨⟩ struct bindings⟨Value⟩  { CM(0,Value::value); };
template ⟨⟩ struct bindings⟨Plus⟩   { CM(0,Plus::exp1);
  ...                                 CM(1,Plus::exp2);   };
template ⟨⟩ struct bindings⟨Divide⟩ { CM(0,Divide::exp1);
                                      CM(1,Divide::exp2); };
```

This binding code would be implicitly provided by the compiler had we chosen that implementation strategy.

The syntax is provided without any external tool support. Instead we rely on a few C++0x features [27], template metaprogramming, and macros. It runs about as fast as the OCaml version (§14.2), and, depending on the usage scenario, compiler and underlying hardware, comes close or outperforms the handcrafted C++ code based on the *visitor design pattern* (§14).

1.1 Motivation

The ideas and the library presented here, were motivated by our rather unsatisfactory experiences working with various C++ front-ends and program analysis frameworks [1, 37, 48**?**]. The problem was not in the frameworks per se, but in the fact that we had to use the *visitor design pattern* [20] to inspect, traverse, and elaborate abstract syntax trees of their target languages. We found visitors unsuitable to express our application logic, surprisingly hard to teach students, and slow. We found dynamic casts in many places, often nested, because users wanted to answer simple structural questions without having to resort to visitors. Users preferred shorter, cleaner, and more direct code to visitors, even at a high cost in performance (assuming that the programmer knew the cost). The usage of **dynamic_cast** resembled the use of pattern matching in functional languages to unpack algebraic data types. Thus, our initial goal was to develop a domain-specific library for C++ to express various predicates on tree-like structures as elegantly as is done in functional languages. This grew into a general high-performance pattern-matching library.

The library is the latest in a series of 5 libraries. The earlier versions were superceded because they failed to meet our standards for notation, performance, or generality. Our standard is set by the principle that a fair comparison must be against the gold standard in a field. For example, if we work on a linear algebra library, we must compare to Fortran or one of the industrial C++ libraries, rather than Java or C. For pattern matching we chose optimized OCaml as our standard for closed (compile-time polymorphic) sets of classes and C++ for uses of the visitor pattern. For generality and simplicity of use, we deemed it essential to do both with a uniform syntax.

1.2 Expression Problem

Functional languages allow for the easy addition of new functions on existing data types, but fall short in extending data types themselves (e.g. with new constructors), which requires modifying the source code. Object-oriented languages, on the other hand, make data type extension trivial through inheritance, but the addition of new functions that work on these classes typically requires changes to the class definition. This dilemma was first discussed by Cook [9] and then accentuated by Wadler [61] under the name *expression problem*. Quoting Wadler:

"The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts)".

To better understand the problem, note that classes differ from algebraic data types in two important ways: they are *extensible* since new variants can be added by inheriting from the base class, as well as *hierarchical* and thus *non-disjoint* since variants can be inherited from other variants and form a subtyping relation between themselves [25]. This is not the case with traditional algebraic data types in functional languages, where the set of variants is *closed*, while the variants are *disjoint*. Some functional languages e.g. ML2000 [2] and Moby [**?**] were experimenting with *hierarchical extensible sum types*, which are closer to object-oriented classes then algebraic data types are, but, interestingly, they did not provide pattern matching facilities on them!

Zenger and Odersky later refined the expression problem in the context of independently extensible solutions [65] as a challenge to find an implementation technique that satisfies the following requirements:

- *Extensibility in both dimensions*: It should be possible to add new data variants, while adapting the existing operations accordingly. It should also be possible to introduce new functions.
- *Strong static type safety*: It should be impossible to apply a function to a data variant, which it cannot handle.
- *No modification or duplication*: Existing code should neither be modified nor duplicated.
- *Separate compilation*: Neither datatype extensions nor addition of new functions should require re-typechecking the original datatype or existing functions. No safety checks should be deferred until link or runtime.
- *Independent extensibility*: It should be possible to combine independently developed extensions so that they can be used jointly.

Object-oriented languages further complicate the matter with the fact that data variants are not necessarily disjoint and may form subtyping relationships between themselves. We thus introduced an additional requirement based on the Liskov substitution principle [34]:

- *Substitutability*: Operations expressed on more general data variants should be applicable to more specific ones that are in a subtyping relation with them.

We will refer to a solution that satisfies all of the above requirements as *open*. Numerous solutions have been proposed to dealing with the expression problem in both functional and object-oriented camps, but notably very few are truly open, while none has made its way into one of the mainstream languages. We refer the reader to Zenger and Odersky's original manuscript for a discussion of the approaches [65]. Interestingly, most of the discussed object-oriented solutions were focusing on the visitor design pattern [20], which even today seems to be the most commonly used approach to dealing with the expression problem in practice.

1.3 Visitor Design Pattern

The *visitor design pattern* [20] was devised to solve the problem of extending existing classes with new functions in object-oriented languages. Consider the above Expr example and imagine that in addition to evaluation we would like to also provide a pretty printing of expressions. A typical object-oriented approach would be to introduce a virtual function
virtual void print() **const** = 0; inside the abstract base class Expr, which will be implemented correspondingly in all the derived classes. This works well as long as we know all the required operations on the abstract class in advance. Unfortunately, this is very difficult to achieve in reality as the code evolves, especially in a production environment. To put this in context, imagine that after the above interface with pretty-printing functionality has been deployed, we decided that we need similar functionality that saves the expression in XML format. Adding new virtual function implies modifying the base class and creating a versioning problem with the code that has been deployed already using the old interface.

To alleviate this problem, the Visitor Design Pattern separates the *commonality* of all such future member-functions from their *specifics*. The former deals with identifying the most-specific derived class of the receiver object known to the system at the time the base class was designed. The latter provides implementation of the required functionality once the most-specific derived class has been identified. The interaction between the two is encoded in the protocol that fixes a *visitation interface* enumerating all known derived classes on one side and a dispatching mechanism that guarantees to select the most-specific case with respect to the dynamic type of the receiver in the visitation interface. An implementation of this protocol for our Expr example might look like the following:

```
// Forward declaration of known derived classes
struct Value; struct Plus; ... struct Divide;

// Visitation interface
struct ExprVisitor
{
    virtual void visit(const Value&) = 0;
    virtual void visit(const Plus&) = 0;
    ... // One virtual function per each known derived class
    virtual void visit(const Divide&) = 0;
};
// Abstract base and known derived classes
struct Expr {
    virtual void accept(ExprVisitor&) const = 0; };
struct Value : Expr { ...
    void accept(ExprVisitor& v) const { v.visit(*this); } };
struct Plus : Expr { ...
    void accept(ExprVisitor& v) const { v.visit(*this); } };
```

Note that even though implementations of accept member-functions in all derived classes are syntactically identical, a different visit is called. We rely here on the overload resolution mechanism of C++ to pick the most specialized visit member-function applicable to the static type of ∗**this**.

A user can now implement new functions by overriding ExprVisitor's functions. For example:

```
std::string to_str(const Expr* e) // Converts expressions to string
{
  struct ToStrVisitor : ExprVisitor
  {
    void visit(const Value& e) { result = std::to_string(e.value); }
    ...
    void visit(const Divide& e) {
        result = to_str(e.exp1) + '/' + to_str(e.exp2);
    }
    std::string result;
  } v;
  e→ accept(v);
  return v.result;
}
```

The function eval we presented above, as well as any new function that we would like to add to Expr, can now be implemented in much the same way, without the need to change the base interface. This flexibility does not come for free, though, and we would like to point out some pros and cons of this solution.

The most important advantage of the visitor design pattern is the **possibility to add new operations** to the class hierarchy without the need to change the interface. Its second most-quoted advantage is **speed** – the overhead of two virtual function calls incurred by the double dispatch present in the visitor design pattern is often negligible on modern architectures. Yet another advantage that often remains unnoticed is that the above solution achieves extensibility of functions with **library only means** by using facilities already present in the language. Nevertheless, there are quite a few disadvantages.

The solution is **intrusive** since we had to inject syntactically the same definition of the accept method into every class participating in visitation. It is also **specific to hierarchy**, as we had to declare a visitation interface specific to the base class. The amount of **boilerplate code** required by visitor design pattern cannot go unnoticed. It also increases with every argument that has to be passed into the visitor to be available during the visitation. This aspect can be seen in the example from §1.4 where we have to store both functors inside the visitor.

More importantly, visitors **hinder extensibility** of the class hierarchy: new classes added to the hierarchy after the visitation interface has been fixed will be treated as their most derived base class present in the interface. A solution to this problem has been proposed in the form of *Extensible Visitors with Default Cases* [64, §4.2]; however, the solution, after remapping it onto C++, has problems of its own, discussed in detail in related work in §15.

Once all the boilerplate related to visitors has been written and the visitation interface has been fixed we are still left with some annoyances incurred by the pattern. One of them is the necessity to work with the **control inversion** that visitors put in place. Because of it we have to save any local state and any arguments that some of the visit callbacks might need from the calling environment. Similarly, we have to save the result of the visitation, as we cannot assume that all the visitors that will potentially be implemented on a given hierarchy will use the same result type. Using visitors in a generic algorithm requires even more precautions. We summarize these visitor-related issues in the following motivating example, followed by an illustration of a pattern-matching solution to the same problem enabled with our library.

1.4 Motivating Example

While comparing generic programming facilities available to functional and imperative languages (mainly Haskell and C++), Dos

Reis and Järvi present the following example in Haskell describing a sum functor[14]:

```
data Either a b = Left a | Right b

eitherLift :: (a → c) → (b → d) → Either a b → Either c d
eitherLift f g (Left x) = Left (f x)
eitherLift f g (Right y) = Right (g y)
```

In simple words, the function eitherLift above takes two functions and an object and depending on the actual type constructor the object was created with, calls first or second function on the embedded value, encoding the result correspondingly.

Its equivalent in C++ is not straightforward. Idiomatic, type-safe, handling of discriminated unions in C++ typically assumes use of the *Visitor Design Pattern*[20].

```
template ⟨class X, class Y⟩ class Either;
template ⟨class X, class Y⟩ class Left;
template ⟨class X, class Y⟩ class Right;

template ⟨class X, class Y⟩
struct EitherVisitor {
    virtual void visit(const Left⟨X,Y⟩&) = 0;
    virtual void visit(const Right⟨X,Y⟩&) = 0;
};

template ⟨class X, class Y⟩
struct Either {
    virtual ~Either() {}
    virtual void accept(EitherVisitor⟨X,Y⟩& v) const = 0;
};

template ⟨class X, class Y⟩
struct Left : Either⟨X,Y⟩ {
    const X& x;
    Left(const X& x) : x(x) {}
    void accept(EitherVisitor⟨X,Y⟩& v) const { v.visit(*this); }
};

template ⟨class X, class Y⟩
struct Right : Either⟨X,Y⟩ {
    const Y& y;
    Right(const Y& y) : y(y) {}
    void accept(EitherVisitor⟨X,Y⟩& v) const { v.visit(*this); }
};
```

The code above defines the necessary parameterized data structures as well as a correspondingly parameterized visitor class capable of introspecting it at run-time. The authors agree with us *"The code has a fair amount of boilerplate to simulate pattern matching..."*[14] The actual implementation of lift in C++ now amounts to declaring and invoking a visitor:

```
template ⟨class X, class Y, class S, class T⟩
const Either⟨S,T⟩& eitherLift(const Either⟨X,Y⟩& e, S f(X), T g(Y))
{
    typedef S (*F)(X);
    typedef T (*G)(Y);
    struct Impl : EitherVisitor⟨X,Y⟩ {
        F f;
        G g;
        const Either⟨S,T⟩* value;
        Impl(F f, G g) : f(f), g(g), value() {}
        void visit(const Left⟨X,Y⟩& e) {
            value = left⟨S,T⟩(f(e.x));
        }
        void visit(const Right⟨X,Y⟩& e) {
            value = right⟨S,T⟩(g(e.y));
        }
    };
    Impl vis(f, g);
    e.accept(vis);
    return *vis.value;
}
```

The same function expressed with our pattern-matching facility seems to be much closer to the original Haskell definition:

```
template ⟨class X, class Y, class S, class T⟩
const Either ⟨S,T⟩* lift (const Either ⟨X,Y⟩& e, S f(X), T g(Y))
{
    Match(e)
      Case(( Left ⟨X,Y⟩), x) return left ⟨S,T⟩(f(x));
      Case((Right⟨X,Y⟩), y) return right ⟨S,T⟩(g(y));
    EndMatch
}
```

This is also as fast as the visitor solution, but unlike the visitors based approach it neither requires EitherVisitor, nor any of the injected accept member-functions. We do require binding definitions though to be able to bind variables x and y:

```
template ⟨class X, class Y⟩
    struct bindings ⟨Left ⟨X,Y⟩⟩ { CM(0, Left⟨X,Y⟩::x); };
template ⟨class X, class Y⟩
    struct bindings ⟨Right⟨X,Y⟩⟩ { CM(0,Right⟨X,Y⟩::y); };
```

Note that these binding definitions are made once for all possible instantiations with the use of partial template specialization in C++ and would not be needed if we implemented pattern matching in a compiler rather than a library.

1.5 Summary

The contributions of the paper are twofold and can be summarized as following:

- We present techniques based on memoization (§9) and class precedence list (§10) that can be used to implement type switching efficiently based on the run-time type of the argument.
  - The techniques come close and often outperform its de facto contender – visitor design pattern – without sacrificing extensibility (§14).
  - They work in the presence of multiple inheritance, including repeated and virtual inheritance, as well as in generic code (§9.3).
  - The solution is open by construction (§8.2), non-intrusive, and avoids the control inversion typical for visitors.
  - It applies to polymorphic (§9.1-9.3) and tagged (§10) class hierarchies through a unified syntax [**?**].
  - Our memoization device (§9.2) generalizes to other languages and can be used to implement type switching (§9.3), type testing (§8.2,[3, §4.7]), predicate dispatch (§9.2), and multiple dispatch (§17) efficiently.
  - We list conditions under which virtual table pointers, commonly used in C++ implementations, uniquely identify the exact subobject within the most derived type (§9.1).
  - We also build an efficient cache indexing function for virtual table pointers that minimizes the amount of conflicts (§9.3.1,9.3.2,[3, §4.3.5]).
- We present a functional style pattern matching for C++ built as a library employing the above technique. Our solution:
  - Is open, non-intrusive and avoids the control inversion typical for visitors.
  - Can be applied retroactively to any polymorphic or tagged class hierarchy.
  - Provides a unified syntax for various encodings of extensible hierarchycal datatypes in C++.

- Generalizes the controversial n+k patterns by leaving semantic choices to the user.
- Supports a limited form of views.
- Is simpler to use than conventional object-oriented or union-based alternatives.
- Improves performance compared to alternatives in real applications.

Our technique can be used in a compiler and/or library setting to implement facilities that depend on dynamic type or run-time properties of objects: e.g. type switching, type testing, pattern matching, predicate dispatch, multi-methods etc. We also look at different approaches to endoding algebraic data types in C++ and present a unified pattern-matching syntax that works uniformly with all of them. We generalize Haskell's n+k patterns[45] to any invertible operations. Semantics issues that typically accompany n+k pattern are handled transparently by forwarding the problem into the concepts domain, thanks to the fact that we work in a library setting. We also provide support for views in a form that resembles extractors in Scala.

A practical benefit of our solution is that it can be used right away with any compiler with a decent support of C++0x without requiring the installation of any additional tools or preprocessors. The solution is a proof of concept that sets a minimum threshold for the performance, brevity, clarity and usefulness of a language solution for open type switching in C++.

The rest of this paper is structured as following. In Section 2, we present evolution of pattern matching in different languages, presenting informally through example commonly used terminology and semantics of various pattern-matching constructs. Section 3 presents various approaches that are taken in C++ to encoding algebraic data types. Sections 4 and 5 describe the syntax and semantics of our pattern matching facilities. Sections 6 and 7 discuss approach taken by our library in handling generalized n+k patterns and views. Section 8 discusses techniques that makes type switching, used as a back-bone of the match statement, efficient, while section 14 provides its performance evaluation against some common alternatives. Section 15 discusses related work, and section 17 concludes by discussing some future directions and possible improvements.

2. Background

Pattern matching in the context of a programming language was first introduced in a string manipulation language SNOBOL[19]. Its fourth reincarnation SNOBOL4 had patterns as first-class data types providing operations of concatenation and alternation on them[24]. The first reference to a pattern-matching construct that resembles the one found in statically typed functional languages today is usually attributed to Burstall and his work on structural induction[4].

In the context of object-oriented programming, pattern matching has been first explored in Pizza programming language[41]. These efforts have been continued in Scala[42] and together with notable work of Burak Emir on *Object-Oriented Pattern Matching*[17] have resulted in incorporation of pattern matching into the language.

Pattern matching has been closely related to *algebraic data types* and *equational reasoning* since the early days of functional programming. In languages like ML and Haskel an *Algebraic Data Type* is a data type each of whose values is picked from a disjoint sum of (possibly recursive) data types, called *variants*. Each of the variants is marked with a unique symbolic constant called *constructor*, while the set of all constructors of a given type is called *signature*. Constructors provide a convenient way of creating a value of its variant type as well as a way of discriminating its variant type from the algebraic data type through pattern matching.

Algebraic data type expr from Section 1 consists of 5 variants, marked with constructors Value, Plus, Minus, Times and Divide respectively. Constructor Value expects a value of type int during construction, as well as any pattern that admits values of type int during decomposition through pattern matching. Similarly, the other four constructors expect a value of a cartesian product of two expr types during construction, as well as any pattern that would admit a value of such type during decomposition.

Algebraic data types can be parameterized and recursive, as demonstrated by the following Haskell code that defines a binary tree parameterized on type k of keys and type d of data stored in the nodes:

```
data Tree k d = Node k d (Tree k d) (Tree k d) | Leaf
```

Naturally, they can be decomposed in a generic algorithm like the function find below, defined through case analysis on the tree's structure:

```
find :: (Ord k) ⇒ k → Tree k d → Maybe d
find i Leaf = Nothing
find i (Node key item left right) =
    if i ==key
    then Just item
    else
        if i <key
        then find i left
        else find i right
```

The set of values described by such an algebraic data type is defined inductively as the least set closed under constructor functions of its variants. Algebraic data types draw their name from the practice of using case distinction in mathematical function definitions and proofs that involve *algebraic terms*.

One of the main differences of algebraic data types from classes in object-oriented languages is that an algebraic data type definition is *closed* because it fixes the structure of its instances once and for all. Once we have listed all the variants a given algebraic data type may have we cannot extend it with new variants without modifying its definition. This is not the case in object-oriented languages, where classes are *open* to extension through subclassing. Notable exceptions to this restriction in functional community are *polymorphic variants* in OCaml[21] and *open data types* in Haskell[36], which allow addition of new variants later. These extensions, however, are simpler than object-oriented extensions as neither polymorphic variants nor open data types form subtyping relation between themselves: open data types do not introduce any subtyping relation, while the subtyping relation on polymorphic variants is a *semantic subtyping* similar to that of XDuce[26], which is based on the subset relation between values of the type. In either case they maintain the important property that each value of the underlying algebraic data type belongs to exactly one disjoint subset tagged with a constructor. The *nominative subtyping* of object-oriented languages does not usually have this disjointness making classes effectively have multiple types. In particular, the case of disjoint constructors can be seen as a degenerated case of a flat class hierarchy among the multitude of possible class hierarchies.

Closedness of algebraic data types is particularly useful for reasoning about programs by case analysis and allows the compiler to perform an automatic *incompleteness* check – test of whether a given *match statement* covers all possible cases. Similar reasoning about programs involving extensible data types is more involved as we are dealing with potentially open set of variants. *Completness* check in such scenario reduces to checking presence of a case that handles the static type of the subject. Absence of such a case, however, does not necessarily imply incompleteness, only potential incompleteness, as the answer will depend on the actual set of variants available at run-time.

A related notion of *redundancy* checking arises from the tradition of using *first-fit* strategy in pattern matching. It warns the user of any *case clause* inside a match statement that will never be entered because of a preceding one being more general. Object-oriented languages, especially C++, typically prefer *best-fit* strategy (e.g. for overload resolution and class template specialization) because it is not prone to errors where semantics of a statement might change depending on the ordering of preceding definitions. The notable exception in C++ semantics that prefers the *first-fit* strategy is ordering of **catch** handlers of a **try**-block. Similarly to functional languages the C++ compiler will perform *redundancy* checking on catch handlers and issue a warning that lists the redundant cases. We use this property of the C++ type system to perform redundancy checking of our match statements in §11.1.

The patterns that work with algebraic data types we have seen so far are generally called *tree patterns* or *constructor patterns*. Their analog in object-oriented languages is often referred to as *type pattern* since it may involve type testing and type casting. Special cases of these patterns are *list patterns* and *tuple patterns*. The former lets one split a list into a sequence of elements in its beginning and a tail with the help of list constructor : and an empty list constructor [] e.g. [x:y:rest]. The latter does the same with tuples using tuple constructor (,,...,) e.g. ([x:xs],'b',(1,2.0),"hi",**True**).

Pattern matching is not used solely with algebraic data types and can equally well be applied to built-in types. The following Haskell code defines factorial function in the form of equations:

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Here 0 in the left hand side of the first *equation* is an example of a *value pattern* (also known as *constant pattern*) that will only match when the actual argument passed to the function factorial is 0. The *variable pattern* n (also referred to as *identifier pattern*) in the left hand side of the second equation will match any value, *binding* variable n to that value in the right hand side of equation. Similarly to variable pattern, the *wildcard pattern* _ will match any value, neither binding it to a variable nor even obtaining it. Value patterns, variable patterns and wildcard patterns are generally called *primitive patterns*. Patterns like variable and wildcard patterns that never fail to match are called *irrefutable*, in contrast to *refutable* patterns like value patterns, which may fail to match.

In Haskell 98[28] the above definition of factorial could also be written as:

```
factorial 0 = 1
factorial (n+1) = (n+1) * factorial n
```

The (n+1) pattern in the left hand side of equation is an example of *n+k pattern*. According to its informal semantics "Matching an $n + k$ pattern (where n is a variable and k is a positive integer literal) against a value v succeeds if $v \geq k$, resulting in the binding of n to $v - k$, and fails otherwise"[45]. n+k patterns were introduced into Haskel to let users express inductive functions on natural numbers in much the same way as functions defined through case analysis on algebraic data types. Besides succinct notation, such language feature could facilitate automatic proof of termination of such functions by compiler. Peano numbers, used as an analogy to algebraic data type representation of natural numbers, is not always the best abstraction for representing other mathematical operations however. This, together with numerous ways of defining semantics of generalized n+k patterns were some of the reasons why the feature was never generalized in Haskell to other kinds of expressions, even though there were plenty of known applications. Moreover, numerous debates over semantics and usefulness of the feature resulted in n+k patterns being removed from the language altogether in Haskell 2010 standard[16]. Generalization of n+k patterns, called *application patterns* has been studied by Nikolaas N. Oosterhof in his Master's thesis[44]. Application patterns essentially treat n+k patterns as equations, while matching against them attempts to solve or validate the equation.

While n+k patterns were something very few languages had, another common feature of many programming languages with pattern matching are guards. 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*. The following OCaml code for exp language from Section 1 defines the rules for factorizing expressions $e_1e_2+e_1e_3$ into $e_1(e_2+e_3)$ and $e_1e_2+e_3e_2$ into $(e_1 + e_3)e_2$ with the help of guards spelled out after keyword when:

```
let factorize  e =
    match e with
      Plus(Times(e1,e2), Times(e3,e4)) when e1 = e3
          → Times(e1, Plus(e2,e4))
    | Plus(Times(e1,e2), Times(e3,e4)) when e2 = e4
          → Times(Plus(e1,e3), e4)
    |   e → e
    ;;
```

One may wonder why we could not simply write the above case clause as Plus(Times(e,e_2), Times(e,e_4)) to avoid the guard? Patterns that permit use of the same variable in them multiple times are called *equivalence patterns*, while the requirement of absence of such patterns in a language is called *linearity*. Neither OCaml nor Haskell support such patterns, while Miranda[57] as well as Tom's pattern matching extension to C, Java and Eiffel[40] supports *non-linear patterns*.

The example above illustrates yet another common pattern-matching facility – *nesting of patterns*. In general, a constructor pattern composed of a linear vector of (distinct) variables is called a *simple pattern*. The same pattern composed not only of variables is called *nested pattern*. Using nested patterns, with a simple expression in the case clause we could define a predicate that tests the top-level expression to be tagged with a Plus constructor, while both of its arguments to be marked with Times constructor, binding their arguments (or potentially pattern matching further) respectively. Note that the visitor design pattern does not provide this level of flexibility and each of the nested tests might have required a new visitor to be written. Nesting of patterns like the one above is typically where users resort to *type tests* and *type casts* that in case of C++ can be combined into a single call to **dynamic_cast**.

Related to nested patterns are *as-patterns* that help one take a value apart while still maintaining its integrity. The following rule could have been a part of a hypothetical rewriting system in OCaml similar to the one above. Its intention is to rewrite expressions of the form $\frac{e_1/e_2}{e_3/e_4}$ into $\frac{e_1}{e_2}\frac{e_4}{e_3} \wedge e_2 \neq 0 \wedge e_3 \neq 0 \wedge e_4 \neq 0$.

```
| Divide(Divide(_,e2) as x, Divide(e3,e4))
      → Times(x, Divide(e4, e3))
```

We introduced a name "x" as a synonym of the result of matching the entire sub-expression Divide(_,e_2) in order to refer it without recomposing in the right-hand side of the case clause. We omitted the conjunction of relevant non-zero checks for brevity, one can see that we will need access to e_2 in it however.

Decomposing algebraic data types through pattern matching has an important drawback that was originally spotted by Wadler[62]: they expose concrete representation of an abstract data type, which conflicts with the principle of *data abstraction*. To overcome the problem he proposed the notion of *views* that represent conversions between different representations that are implicitly applied during pattern matching. As an example, imagine polar and cartesian representations of complex numbers. A user might choose polar representation as a concrete representation for the abstract data type complex, treating cartesian representation as view or vice versa:[1]

```
complex ::= Pole real real
view complex ::= Cart real real
  in  (Pole r t) = Cart (r * cos t) (r * sin t)
  out (Cart x y) = Pole (sqrt(x^2 + y^2)) (atan2 x y)
```

The operations then might be implemented in whatever representation is the most suitable, while the compiler will implicitly convert representation if needed:

```
  add  (Cart x1 y1) (Cart x2 y2) = Cart (x1 + x2) (y1 + y2)
  mult (Pole r1 t1) (Pole r2 t2) = Pole (r1 * r2) (t1 + t2)
```

The idea of views were later adopted in various forms in several languages: Haskell[6], Standard ML[43], Scala (in the form of *extractors*[17]) and F♯ (under the name of *active patterns*[13]).

Logic programming languages like Prolog take pattern matching to even greater level. The main difference between pattern matching in logic languages and functional languages is that functional pattern matching is a "one-way" matching where patterns are matched against values, possibly binding some variables in the pattern along the way. Pattern matching in logic programming is "two-way" matching based on *unification* where patterns can be matched against other patterns, possibly binding some variables in both patterns and potentially leaving some variables *unbound* or partially bound – i.e. bound to patterns. A hypothetical example of such functionality can be matching a pattern Plus(x,Times(x,1)) against another pattern Plus(Divide(y,2),z), which will result in binding x to a Divide(y,2) and z to Times(Divide(y,2),1) with y left unbound, leaving both x and z effectively a pattern.

2.1 Expression Templates

Interestingly enough C++ has a pure functional sublanguage in it that has a striking similarity to ML and Haskell. The sublanguage in question is template facilities of C++ that has been shown to be Turing complete[60].

Haskell definition of factorial we saw earlier can be rewritten in template sublanguage of C++ as following:

```
template ⟨int N⟩
    struct factorial { enum { result = N*factorial⟨N−1⟩::result }; };
template ⟨⟩
    struct factorial⟨0⟩ { enum { result = 1 }; };
```

One can easily see similarity with equational definitions in Haskell, with the exception that more specific cases (specialization for 0) have to follow the general definition in C++. The main difference between Haskell definition and its C++ counterpart is that the former describes computations on *run-time values*, while the latter can only work with *compile-time values*.

Turns out we can even express our *exp* language using this functional sublanguage:

```
template ⟨class T⟩
struct value {
    value(const T& t) : m_value(t) {}
    T m_value;
};

template ⟨class T⟩
struct variable {
    variable() : m_value() {}
    T m_value;
};

template ⟨typename E1, typename E2⟩
struct plus {
    plus(const E1& e1, const E2& e2) : m_e1(e1), m_e2(e2) {}
    const E1 m_e1; const E2 m_e2;
};

// ... definitions of other expressions
```

The idea is that expressions can be composed out of subexpressions, whose shape (type) is passed as arguments to above templates. Explicit description of such expressions is very tedious however and is thus never expressed directly, but as a result of corresponding operations:

```
template ⟨typename T⟩
    value⟨T⟩ val(const T& t) { return value⟨T⟩(t); }
template ⟨typename E1, typename E2⟩
    plus⟨E1,E2⟩ operator+(const E1& e1, const E2& e2)
    { return plus⟨E1,E2⟩(e1,e2); }
```

With this, one can now capture various expressions as following:

```
variable⟨int⟩ v;
auto x = v + val(3);
```

The type of variable x – plus⟨variable⟨int⟩,value⟨int⟩⟩ – captures the structure of the expression, while the values inside of it represent various subexpressions the expression was created with. Such an expression can be arbitrarily, but finitely nested. Note that value 3 is not added to the value of variable v here, but the expression v+3 is recorded, while the meaning to such expression can be given differently in different contexts. A general observation is that only the shape of the expression becomes fixed at compile time, while the values of variables involved in it can be changed arbitrarily at run time, allowing for *lazy evaluation* of the expression. Polymorphic function eval below implements just that:

```
template ⟨typename T⟩
    T eval(const value⟨T⟩& e) { return e.m_value; }
template ⟨typename T⟩
    T eval(const variable ⟨T⟩& e) { return e.m_value; }
template ⟨typename E1, typename E2⟩
    auto eval(const plus⟨E1,E2⟩& e)
        → decltype(eval(e.m_e1) + eval(e.m_e2))
          { return eval(e.m_e1) + eval(e.m_e2); }
```

One can now modify value of the variable v and re-evaluate expression as following:

```
v = 7; // assumes overloading of assignment
int r = eval(x); // returns 10
```

The above technique for lazy evaluation of expressions was independently invented by Todd Veldhuizen and David Vandevoorde and is generally known in the C++ community by the name *Expression Templates* that Todd coined[58, 59].

Note again how implementation of eval resembles equations in Haskell that decompose an algebraic data type. The similarities are so striking that there were attempts to use Haskell as a pseudo code language for template metaprogramming in C++[38]. A key observation in this analogy is that partial and explicit template specialization of C++ class templates are similar to defining equations for Haskell functions. Variables introduced via template clause of each

[1] We use syntax from Wadler's original paper for this example

equation serve as *variable patterns*, while the names of actual templates describing arguments serve as *variant constructors*. An important difference between the two is that Haskell's equations use *first-fit* strategy making order of equations important, while C++ uses *best-fit* strategy, thus making the order irrelevant.

Patterns expressed this way can be arbitrarily nested as long as they can be expressed in terms of the types involved and not the values they store. Using the above example, for instance, it is very easy to specialize eval for an expression of form $c_1 * x + c_2$ where c_i are some (not known) constant values and x is any variable. Specializing for a concrete instance of that expression $2 * x + 3$ will be much harder, because in the representation we chose values 2 and 3 become run-time values and thus cannot participate in compile-time computations anymore. In this case we could have devised a template that allocates a dedicated type for each constant making such value part of the type:

```
template ⟨class T, T t⟩ struct constant {};

template ⟨typename T, T t⟩
    T eval(const constant⟨T,t⟩& e) { return t; }
template ⟨typename E⟩
    auto eval(const times⟨constant⟨int,0⟩, E⟩& e)
        → decltype(eval(e.m_e2))
            { return (decltype(eval(e.m_e2)))(0); }
template ⟨typename E⟩
    auto eval(const times⟨E,constant⟨int,0⟩⟩& e)
        → decltype(eval(e.m_e1))
            { return (decltype(eval(e.m_e1)))(0); }
```

Here the first equation for eval describes the necessary general case for handling expressions of type constant⟨T,t⟩, while the other two are redundant cases that can be seen as an optimization detecting expressions of the form $e * 0$ and $0 * e$ for any arbitrary expression e and returning 0 without actually computing e.

Unfortunately, a similar pattern to detect expressions of the form $x - x$ for any variable x cannot be expressed because expression templates are blind to object identity and can only see their types. This means that expression templates of the form $x - y$ are indisthinguishable at compile time from expressions of the form $x - x$ because their types are identical.

Nevertheless, with all the limitations, expression templates provide an extremely powerful abstraction mechanism, which we use to express a pattern-language for our SELL. Coincidentally, we employ the compile-time pattern-matching facility already supported by C++ as a meta-language to implement its run-time counterpart.

3. Algebraic Data Types in C++

C++ does not have a direct support of algebraic data types, but they can usually be emulated in a number of ways. A pattern-matching solution that strives to be general will have to account for different encodings and be applicable to all of them.

Consider an ML data type of the form:

$$\textbf{datatype}\ DT = C_1\ \textbf{of}\ \{L_{11} : T_{11}, ..., L_{1m} : T_{1m}\} \\ \quad |\ ... \\ \quad |\ C_k\ \textbf{of}\ \{L_{k1} : T_{k1}, ..., L_{kn} : T_{kn}\}$$

There are at least 3 different ways to represent it in C++. Following Emir, we will refer to them as *encodings* [17]:

- Polymorphic Base Class (or *polymorphic encoding* for short)
- Tagged Class (or *tagged encoding* for short)
- Discriminated Union (or *union encoding* for short)

In polymorphic and tagged encoding, base class DT represents algebraic data type, while derived classes represent variants. The only difference between the two is that in polymorphic encoding base class has virtual functions, while in tagged encoding it has a dedicated member of integral type that uniquely identifies the variant – derived class.

The first two encodings are inherently *open* because the classes can be arbitrarily extended through subclassing. The last encoding is inherently *closed* because we cannot add more members to the union without modifying its definition.

When we deal with pattern matching, the static type of the original expression we are matching may not necessarily be the same as the type of expression we match it with. We call the original expression a *subject* and its static type – *subject type*. We call the type we are trying to match subject against – a *target type*.

In the simplest case, detecting that the target type is a given type or a type derived from it, is everything we want to know. We refer to such a use-case as *type testing*. In the next simplest case, besides testing we might want to get a pointer or a reference to the target type of subject as casting it to such a type may involve a non-trivial computation only a compiler can safely generate. We refer to such a use-case as *type identification*. Type identification of a given subject against multiple target types is typically referred to as *type switching*.

Once we uncovered the target type, we may want to be able to decompose it *structurally* (when the target type is a *structured* data type like array, tuple or class) or *algebraically* (when the target type is a scalar data type like **int** or **double**). Structural decomposition in our library can be performed with the help of *tree patterns*, while algebraic decomposition can be done with the help of *generalized n+k patterns*.

3.1 Polymorphic Base Class

In this encoding user declares a polymorphic base class DT that will be extended by classes representing all the variants. Base class might declare several virtual functions that will be overridden by derived classes, for example accept used in a Visitor Design Pattern.

```
class DT { virtual ~DT{} };
class C_1 : public DT {T_11 L_11; ...T_1m L_1m;}
...
class C_k : public DT {T_k1 L_k1; ...T_kn L_kn;}
```

The uncover the actual variant of such an algebraic data type, the user might use **dynamic_cast** to query one of the k expected run-time types (an approach used by Rose[52]) or she might employ a visitor design pattern devised for this algebraic data type (an approach used by Pivot[48] and Phoenix[37]). The most attractive feature of this approach is that it is truly open as we can extend classes arbitrarily at will (leaving the orthogonal issues of visitors aside).

3.2 Tagged Class

This encoding is similar to the *Polymorphic Base Class* in that we use derived classes to encode the variants. The main difference is that the user designates a member in the base class, whose value will uniquely determine the most derived class a given object is an instance of. Constructors of each variant C_i are responsible for properly initializing the dedicated member with a unique value c_i associated with that variant. Clang[1] among others uses this approach.

```
class DT { enum kinds {c_1, ..., c_k} m_kind; };
class C_1 : public DT {T_11 L_11; ...T_1m L_1m;}
...
class C_k : public DT {T_k1 L_k1; ...T_kn L_kn;}
```

In such scenario the user might use a simple switch statement to uncover the type of the variant combined with a **static_cast** to properly cast the pointer or reference to an object. People might prefer this encoding to the one above for performance reasons as it is possible to avoid virtual dispatch with it altogether. Note, however, that once we allow for extensions and not limit ourselves with encoding algebraic data types only it also has a significant drawback in comparison to the previous approach: we can easily check that given object belongs to the most derived class, but we cannot say much about whether it belongs to one of its base classes. A visitor design pattern can be implemented to take care of this problem, but control inversion that comes along with it will certainly diminish the convenience of having just a switch statement. Besides, forwarding overhead might lose some of the performance benefits gained originally by putting a dedicated member into the base class.

3.3 Discriminated Union

This encoding is popular in projects that are either implemented in C or originated from C before coming to C++. It involves a type that contains a union of its possible variants, discriminated with a dedicated value stored as a part of the structure. The approach is used by EDG front-end[15] and many others.

```
struct DT
{
    enum kinds {c_1, ..., c_k} m_kind;
    union {
        struct C_1 {T_11 L_11; ...T_1m L_1m;} C_1;
        ...
        struct C_k {T_k1 L_k1; ...T_kn L_kn;} C_k;
    };
};
```

As before, the user can use a switch statement to identify the variant c_i and then access its members via C_i union member. This approach is truly closed, as we cannot add new variants to the underlying union without modifying class definition.

Note also that in this case both subject type and target types are the same and we use an integral constant to distinguish which member(s) of the underlying union is active now. In the other two cases the type of a subject is a base class of the target type and we use either run-time type information or the integral constant associated by the user with the target type to uncover the target type.

4. Pattern Matching Syntax

Figure 1 presents the syntax enabled by our SELL in an abstract syntax form rather than traditional EBNF in order to better describe compositions allowed by the library. In particular, the allowed compositions depend on the C++ type of the entities being composed, so we need to include it in the notation. We do make use of several non-terminals from the C++ grammar in order to put the use of our constructs into context.

Match statement is an analog of a switch statement that allows case clauses to be used as its case statements. We require it to be terminated with a dedicated *EndMatch* macro, to properly close the syntactic structure introduced with *Match* and followed by *Case*,Que and *Otherwise* macros. Match statement will accept subjects of pointer and reference types, treating them uniformly in case clauses. This means that user does not have to mention ∗,& or any of the **const**,**volatile**-qualifiers when specifying target types. Passing nullptr as a subject is considered *ill-formed* however – a choice we have made for performance reasons. Examples of match statement has already been presented in §1 and §1.4.

match statement	M	$::=$	$\mathit{Match}(e)\ [Cs^*]^*\ \mathit{EndMatch}$
case clause	C	$::=$	$\mathit{Case}(T\,[,x]^*)$
		$\mid$	$\mathrm{Que}(\,T\,[,\omega]^*)$
		$\mid$	$\mathit{Otherwise}([,x]^*)$
target expression	T	$::=$	$\tau \mid l \mid \nu$
view	ν	$::=$	$\mathrm{view}\langle\tau,l\rangle$
match expression	m	$::=$	$\pi(e)$
pattern	π	$::=$	$\_ \mid \eta \mid \varrho \mid \mu \mid \varsigma \mid \chi$
extended pattern	ω	$::=$	$\pi \mid c \mid x$
tree pattern	μ	$::=$	$\mathrm{match}\langle\nu \mid \tau\,[,l]\rangle(\omega^*)$
guard pattern	ϱ	$::=$	$\pi \vDash \xi$
n+k pattern	η	$::=$	$\chi \mid \eta \oplus c \mid c \oplus \eta \mid \ominus\eta \mid (\eta) \mid \_$
wildcard pattern	$\_^{wildcard}$		
variable pattern	χ	$::=$	$\kappa \mid \iota$
value pattern	$\varsigma^{value\langle\tau\rangle}$		
xt variable	$\kappa^{variable\langle\tau\rangle}$		
xt reference	$\iota^{var\_ref\langle\tau\rangle}$		
xt expression	ξ	$::=$	$\chi \mid \xi \oplus c \mid c \oplus \xi \mid \ominus\xi \mid (\xi) \mid \xi \oplus \xi$
layout	l	$::=$	c^{int}
unary operator	$\ominus$	$\in$	$\{*, \&, +, -, !, \sim\}$
binary operator	$\oplus$	$\in$	$\{*, /, \%, +, -, \ll, \gg, \&, \wedge, \mid,$ $<, \leq, >, \geq, =, \neq, \&\&, \mid\mid\}$
type-id	τ		C++[27, §A.7]
statement	s		C++[27, §A.5]
expression	e^τ		C++[27, §A.4]
constant-expression	c^τ		C++[27, §A.4]
identifier	x^τ		C++[27, §A.2]

Figure 1. Syntax enabled by out pattern-matching library

We support three kinds of **case clauses**: *Case-clause*, Que-*clause* and *Otherwise-clause* also called *default clause*. *Case* and Que clauses are refutable and both take a target expression as their first argument. *Otherwise* clause is irrefutable and can occur at most once among the clauses. Its target type is the subject type. *Case* and *Otherwise* clauses take additionally a list of identifiers that will be treated as variable patterns implicitly introduced into the clause's scope and bound to corresponding members of their target type. Que clause permits nested patterns as its arguments, but naturally requires all the variables used in the patterns to be explicitly pre-declared. Even though our default clause is not required to be the last clause of the match statement, we strongly encourage the user to place it last (hence the choice of name – otherwise). Placing it at the beginning or in the middle of a match statement will only work as expected with *tagged class* and *discriminated union* encodings that use *the-only-fit-or-default* strategy for choosing cases. The *polymorphic base class* encoding uses *first-fit* strategy and thus irrefutable default clause will effectively hide all subsequent case clauses, making them redundant. As we show in §**??** the switch between *polymorphic base class* and *tagged class* encodings can simply be made with addition or removal a single definition, which may inadvertently change semantics of those match statements, where the default clause were not placed last.

When default clause takes optional variable patterns, it behaves in exactly the same way as *Case* clause whose target type is the subject type.

Target expression used by the case clauses can be either a target type, a constant value, representing *layout* (§4.1) or a *view* type combining the two (§7). Constant value is only allowed for union encoding of algebraic data types, in which case the library assumes the target type to be the subject type.

Views in our library are represented by instantiations of a template class view⟨T,l⟩ that takes a target type and a layout, combining the two into a new type. Our library takes care of transparent handling of this new type as the original combination of target type and layout. Views are discussed in details in §7.

Match expression can be seen as an inline version of match statement with a single Que-clause. Once a pattern is created, it can be applied to an expression in order to check whether that expression matches the pattern, possibly binding some variables in it. The result of application is always of type **bool** except for the tree pattern, where it is a value convertible to **bool**. The actual value in this case is going to be a pointer to target type T in case of a successful match and a nullptr otherwise. Match expressions will most commonly be seen to quickly decompose a possibly nested expression with a tree pattern as seen in the example in the paragraph below. They are the most used expressions under the hood however that let our library be composable.

Pattern summarizes *applicative patterns* – patterns that can be used in a match expression described above. For convenience reasons this category is extended with c and x to form an **extended pattern** – a pattern that can be used as an argument of *tree pattern* and Que clause. Extended pattern lets us use constants as a *value pattern* and regular C++ variables as a *variable pattern* inside these constructs. The library implicitly recognizes them and transforms into ς and ι respectively. This transformation is further explained in §5.3 with $\overset{flt}{\vdash}$ rule set.

Tree pattern takes a target type and an optional layout as its template arguments, which uniquely determines a concrete decomposition scheme for the type. Any nested sub-patterns are taken as run-time arguments. Besides applicative patterns, we allow constants and regular C++ variables to be passed as arguments to a tree pattern. They are treated as *value patterns* and *variable patterns* respectively. Tree patterns can be arbitrarily nested. The following example reimplements factorize from §2 in C++ enhanced with our SELL:

```
const Expr* factorize(const Expr* e)
{
    const Expr *e1, *e2, *e3, *e4;
    if (match⟨Plus⟩(match⟨Times⟩(e1,e2),match⟨Times⟩(e3,e4))(e))
        if (e1 ==e3)
            return new Times(e1, new Plus(e2,e4));
        else
        if (e2 ==e4)
            return new Times(new Plus(e1,e3), e4);
    return e;
}
```

The above example instantiates a nested pattern and then immediately applies it to value e to check whether the value matches the pattern. If it does, it binds local variables to sub-expressions, making them available inside if. Examples like this are known to be a week spot of visitor design pattern and we invite you to implement it with it in order to compare both solutions.

Guard patterns in our SELL consist of two expressions separated by operator |=[2]: an expression being matched (left operand) and a condition (right operand). The right operand is allowed to make use of the variable bound in the left operand. When used on arguments of a tree pattern, the condition is also allowed to make use of any variables bound by preceding argument positions. Naturally, a guard pattern that follows a tree pattern may use all the variables bound by the tree pattern. Consider for example decomposition of a color value, represented as a three-byte RGB triplet:

```
variable ⟨double⟩ r, g, b;
auto p = match⟨RGB⟩(
            255*r,
            255*g |= g < r,
            255*b |= b < g+r
         ) |= r+g+b ≤ 0.5;
```

[2] Operator |= defining the guard was chosen arbitrarily from those that have relatively low precedence in C++. This was done to allow most of the other operators be used inside the condition without parenthesis

Note that C++ standard leaves the order of evaluation of functions arguments unspecified[27, §8.3.6], while we seem to rely here on *left-to-right* order. The reason we can do this lays in the fact that for the purpose of pattern matching all the sub-expressions are evaluated lazily and the unspecified order of evaluation refers only to the order in which corresponding expression templates are created. The actual evaluation of these expression templates happens later when the pattern is applied to an expression and since at that point we have an entirely built expression at hand, we ourselves enforce its evaluation order.

Another important bit about our guards that has to be kept in mind is that guards depend on lazy evaluation and thus expression templates. This is why the variables mentioned inside a guard pattern must be of type variable⟨T⟩ and not just T. Failing to declare them as such will result in eager evaluation of guard expression as a normal C++ expression. This will usually go unnoticed at compile time, while surprising at run time, especially to novices.

We chose to provide a syntax for guards to be specified on arguments of a tree pattern in addition to after the pattern in order to detect mismatches early without having to compute and either bind or match subsequent arguments.

n+k patterns are essentially a subset of *xt expressions* with at most one non-repeated variable in them. This allows for expressions like $2x + 1$, but not for $x + x + 1$, which even though is semantically equivalent to the first one, will not be accepted by our library as an *n+k pattern*.

Expressions 255*r, 255*g and 255*b in the example above were instances of our *generalized n+k patterns*. Informally they meant the following: the value we are matching against is of the form $255 * x$, what is the value of x? Since color components were assumed to be byte values in the range $[0 - 255]$ the user effectively gets normalized RGB coordinates in variables r, g and b ranging over $[0.0 - 1.0]$.

n+k pattern are visually appealing in the sense that they let us write code very close to mathematical notations often used in literature. Consider the definition of fast Fibonacci algorithm taken almost verbatim from the book. Function sqr here returns a square of its argument.

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

Wildcard pattern in our library is represented by a predefined global variable _ of a dedicated type wildcard bearing no state. Wildcard pattern is accepted everywhere where a *variable pattern* χ is accepted. The important difference from a use of an unused variable is that no code is executed to obtain a value for a given position and copy that value into a variable. The position is ignored altogether and the pattern matching continues.

There are two kinds of **variable patterns** in our library: *xt variable* and *xt reference*. **xt variable** stands for *expression-template variable* and refers to variables whose type is variable⟨T⟩ for any given type T. **xt reference** is an *expression-template variable ref-*

erence whose type is $\mathsf{var\_ref}\langle\mathsf{T}\rangle$. The latter is never declared explicitly, but is implicitly introduced by the library to wrap regular variables in places where our syntax accepts x. Both variable kinds are terminal symbols in our SELL letting us build expression templates of them. The main difference between the two is that **xt variable** κ maintains a value of type T as its own state, while **xt reference** ι only keeps a reference to a user-declared variable of type T. Besides the difference in where the state is stored, both kinds of variables have the same semantics and we will thus refer to them as χ. We will also use χ^τ to mean either $\kappa^{variable\langle\tau\rangle}$ or $\iota^{var\_ref\langle\tau\rangle}$ since the type of the actual data τ is what we are interested in, while the fact that it was wrapped into variable⟨⟩ or var_ref⟨⟩ can be implicitly inferred from the meta-variable χ. We would also like to point out that in most cases a variable pattern can be used as regular variable in the same context where a variable of type T can. In few cases where this does not happen implicitly, the user might need to put an explicit cast to T& or **const** T&.

Value pattern is similarly never declared explicitly and is implicitly introduced by the library in places where c is accepted.

xt expression refers to an *expression-template expression* – a non-terminal symbol in our expression language built by applying a given operator to argument expression templates. We use this syntactic category to distinguish lazily evaluated expressions introduced by our SELL from eagerly evaluated expressions, directly supported by C++.

Layout is an enumerator that user can use to define alternative bindings for the same class. When layout is not mentioned, the default layout is used, which is the only required layout a user has to define if she wishes to make use of bindings. We discuss layouts in details in §4.1.

Binary operator and **unary operator** name a subset of C++ operators we make use of and provide support for in our pattern-matching library.

The remaining syntactic categories refer to non-terminals in the C++ grammar bearing the same name. **Identifier** will only refer to variable names in our SELL, even though it has a broader meaning in the C++ grammar. **Expression** subsumes any valid C++ expression. We use expression e^τ to refer to a C++ expression, whose result type is τ. **Constant-expression** is a subset of the above restricted to only expression computable at compile time. **Statement** refers to any valid statement allowed by the C++ grammar. Our match statement M would have been extending this grammar rule with an extra case should it have been defined in the grammar directly. **Type-id** represents a type expression that designates any valid C++ type. We are using this meta-variable in the superscript to other meta-variables in order to indicate a C++ type of the entity they represent.

4.1 Bindings Syntax

Structural decomposition in functional languages is done with the help of constructor symbol and a list of patterns in positions that correspond to arguments of that constructor. C++ allows for multiple constructors in a class, often overloaded for different types but the same arity. Implicit nature of variable patterns that matches "any value" will thus not help in disambiguating such constructors, unless the user explicitly declares the variables, thus fixing their types. Besides, C++ does not have means for general compile-time reflection, so a library like ours cannot enumerate all the constructors present in a class automatically. This is why we decided to separate *construction* of objects from their *decomposition* through pattern matching with *bindings*.

bindings		::=	δ^*
binding definition	δ	::=	**template** $\langle[\vec{p}]\rangle$ **struct** bindings$\langle\ \tau[\langle\vec{p}\rangle]\,[,l]\rangle$ $\{\ [ks]\,[kv]\,[bc]\,[cm^*]\ \};$
class member	cm	::=	$\mathit{CM}(c^{size\_t}, q);$
kind selector	ks	::=	$\mathit{KS}(q);$
kind value	kv	::=	$\mathit{KV}(c);$
base class	bc	::=	$\mathsf{BC}(\tau);$
template-parameter-list	$\vec{p}$		C++[**?** , §A.12]
qualified-id	q		C++[**?** , §A.4]

Figure 2. Syntax used to provide bindings to concrete class hierarchy

The following grammar defines a syntax for a sublanguage our user will use to specify decomposition of various classes for pattern matching:[3]

Any type τ may have arbitrary amount of *bindings* associated with it and distinguished through the *layout* parameter l. The *default binding* which omits layout papameter is implicitly associated with layout whose value is equal to predefined constant default_layout = size_t(~0). User-defined layouts should not reuse this dedicated value.

A *Binding definition* consists of either full or partial specialization of a template-class:

```
template ⟨typename T, size_t l = default_layout⟩
    struct bindings;
```

The body of the class consists a sequence of specifiers, which generate the necessary definitions for querying bindings by the library code. 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.

A *Class Member* specifier $\mathit{CM}(c, q)$ that takes (zero-based) binding position c and a qualified identifier q, specifies a member, whose value will be used to bind variable in position c of τ's decomposition with this *binding definition*. Qualified identifier is allowed to be of one of the following kinds:

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

The following example definition provides bindings to the standard library type std::complex⟨T⟩:

```
template ⟨typename T⟩ struct bindings⟨std::complex⟨T⟩⟩ {
    CM(0, std::complex⟨T⟩::real );
    CM(1, std::complex⟨T⟩::imag);
};
```

It states that when pattern matching against std::complex⟨T⟩ for any given type T, use the result of invoking member-function real() to obtain the value for the first pattern matching position and imag() for the second position.

In the presence of several overloaded members with the same name but different arity, *CM* will unambiguously pick one that falls into one of the three listed above categories of accepted members. In the example above, nullary T std::complex⟨T⟩::real() **const** is preferred to unary **void** std::complex⟨T⟩::real(T).

[3] We reuse several meta-variables introduced in the previous grammar

Note that the binding definition above is made once for all instantiations of std::complex and can be fully or partially specialized for cases of interest. Non-parameterized classes will fully specialize the bindings trait to define their own bindings.

Using *CM* specifier a user defines the semantic functor $\Delta_i^{\tau,l}, i = 1..k$ we introduced in §5.1 as following:

```
template ⟨⟩struct bindings⟨τ⟩ {
    CM(0, τ::member_for_position_0);
    ...
    CM(k, τ::member_for_position_k);
};
```

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.

A *Kind Selector* specifier $KS(q)$ is used to specify a member of the subject type that will uniquely identify the variant for *tagged* and *union* encodings. The member q can be of any of the three categories listed for *CM*, but is required to return an *integral type*.

A *Kind Value* specifier $KV(c)$ is used by *tagged* and *union* encodings to specify a constant c that uniquely identifies the variant.

A *Base Class* specifier $\mathsf{BC}(\tau)$ is used by the *tagged* encoding to specify an immediate base class of the class whose bindings we define. A helper $\mathsf{BCS}(\tau*)$ specifier can also be used to specify the exact topologically sorted list of base classes (§10).

A *Layout* parameter l can be used to define multiple bindings for the same target type. This is particularly essential for *union* encoding where the types of the variants are the same as the type of subject and thus layouts become the only way to associate variants with position bindings. For this reason we require binding definitions for *union* encoding always use the same constant l as a kind value specified with $KV(\mathsf{l})$ and the layout parameter l!

Consider for example the following discriminated union describing various shapes:

```
struct cloc { double first ; double second; };
struct ADTShape
{
    enum shape_kind {circle, square, triangle } kind;
    union {
      struct { cloc center;       double radius; }; // circle
      struct { cloc upper_left ; double size ; };    // square
      struct { cloc first , second, third ; };        // triangle
    };
};

template ⟨⟩ struct bindings⟨ADTShape⟩ {
    KS(ADTShape::kind);     // Kind Selector
};
template ⟨⟩ struct bindings⟨ADTShape, ADTShape::circle⟩ {
    KV(ADTShape::circle);   // Kind Value
    CM(0, ADTShape::center);
    CM(1, ADTShape::radius);
};
```

KS specifier within default bindings for ADTShape tells the library that value of a ADTShape::kind member, extracted from subject at run time, should be used to obtain a unique value that identifies the variant. Binding definition for circle variant then uses the same constant ADTShape::circle as the value of the layout parameter of the bindings⟨T,l⟩ trait and $KV(\mathsf{l})$ specifier to indicate its *kind value*.

Should the shapes have been encoded with a *Tag Class*, the bindings for the base class Shape would have contained *KS*(Shape::kind) specifier, while derived classes Circle, Square and Triangle, representing corresponding variants, would have had *KV*(Shape::circle) etc. specifiers in their binding definitions. These variant classes could have additionally defined a few alternative layouts for themselves, in which case the numbers for the layout parameter could have been arbitrarily chosen.

5. Pattern Matching Semantics

We use natural semantics[29] (big-step operational semantics) to describe our pattern-matching semantics. As with syntax, we do not formalize the semantics of the entire language, but concentrate only on presenting relevant parts of our extension. We assume the entire state of the program is modeled by an environment Γ, which we can query as $\Gamma(x)$ to get a value of a variable x. In addition to metavariables we have seen already, metavariables u, v and b^{bool} range over values. We make a simplifying assumption that all values of user-defined types are represented via variables of reference types and there exist a non-throwing operation $dc\langle\tau\rangle(v)$ that can test whether an object of a given type is an instance of another type, returning a proper reference to it or a dedicated value $\bot$ that represents nullptr. Intuitively, the semantics of such references is that of pointers in C++, which are implicitly dereferenced. We describe our semantics with several rule sets that deal with different parts of our syntax.

5.1 Semantics of Matching Expressions

The rule set in Figure 3 deals with pattern application $\pi(e)$, which essentially performs matching of a pattern π against expression e. The judgements are of the form $\Gamma \vdash \pi(e) \Rightarrow v, \Gamma'$ that can be interpreted as given an environment Γ, pattern application $\pi(e)$ results in value v and environment Γ'. When we use $\Rightarrow^+$ instead of $\Rightarrow$ we simply pointing out that corresponding evaluation rule comes from the C++ semantics and not from our rules.

Matching a wildcard pattern against expression always succeeds without changes to the environment (WILDCARD). Matching a value pattern against expression succeeds only if the result of evaluating that expression is the same as the constant (VALUE). Matching against variable will always succeeds when the type of expression e is the same as variable's value type τ. When the types are different, the library will try to use **dynamic_cast**⟨τ⟩(e) to see whether dynamic type of expression can be casted to τ. If it does, matching succeeds, binding variable to the result of **dynamic_cast**. If it does not, matching fails (VARIABLE).

Our semantics for generalized n+k patterns draws on the notion of *backward collecting semantics* used in *abstract interpretation*[10]. In general, backward collecting semantics $Baexp\,[\![A]\!]\,(E)R$ of an expression A defines the subset of possible environments E such that the expression may evaluate, without producing a runtime error, to a value belonging to given set R:

$$Baexp\,[\![A]\!]\,(E)R = \{x \in E | A(x) \in R\}$$

This can be interpreted as following: given a set E from where the arguments of an expression A draw their values, as well as a set R of acceptable (not all possible) results, find the largest subset $X \subseteq E$ of arguments that will only render results of evaluating A on them in R.

Intuitively n+k patterns like $f(x, y) = v$ relate a known result of a given function application to its arguments (hence analogy with backward collecting semantics). The case where multiple unknown arguments are matched against a single result should not be immediately discarded as there are known n-ary functions whose inverse is unique. An example of such function is Cantor pairing function that defines bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$. Even when such mappings are not one-to-one, their restriction to a given argument often is. Most generalizations of n+k patterns seem to agree on the following rules:

WILDCARD

$$\Gamma \vdash \_(e) \Rightarrow true, \Gamma$$

VALUE

$$\frac{\Gamma \vdash e \Rightarrow^{+} v, \Gamma_1 \qquad \Gamma_1 \overset{eval}{\vdash} \varsigma \Rightarrow u, \Gamma_2}{\Gamma \vdash \varsigma^{\tau}(e^{\tau}) \Rightarrow (u == v), \Gamma_2}$$

VARIABLE

$$\frac{\Gamma \vdash e \Rightarrow^{+} v, \Gamma_1 \qquad \Gamma_1 \vdash dc\langle\tau_1\rangle(v) \Rightarrow^{+} u, \Gamma_2}{\Gamma \vdash \chi^{\tau_1}(e^{\tau_2}) \Rightarrow (u \neq \bot), \Gamma_2[\chi \leftarrow u]}$$

N+K BINARY LEFT

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow^{+} v_1, \Gamma_1\\ \Gamma_1 \vdash \Psi_{\oplus}^{\tau}[v_1](\bullet, c) \Rightarrow^{+} \langle b_2, v_2\rangle, \Gamma_2 \qquad \Gamma_2 \vdash \eta(v_2) \Rightarrow b_3, \Gamma_3\end{array}}{\Gamma \vdash (\eta^{\tau} \oplus c)(e) \Rightarrow (b_2 \wedge b_3), \Gamma_3}$$

N+K BINARY RIGHT

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow^{+} v_1, \Gamma_1\\ \Gamma_1 \vdash \Psi_{\oplus}^{\tau}[v_1](c, \bullet) \Rightarrow^{+} \langle b_2, v_2\rangle, \Gamma_2 \qquad \Gamma_2 \vdash \eta(v_2) \Rightarrow b_3, \Gamma_3\end{array}}{\Gamma \vdash (c \oplus \eta^{\tau})(e) \Rightarrow (b_2 \wedge b_3), \Gamma_3}$$

N+K UNARY

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow^{+} v_1, \Gamma_1\\ \Gamma_1 \vdash \Psi_{\ominus}^{\tau}[v_1](\bullet) \Rightarrow^{+} \langle b_2, v_2\rangle, \Gamma_2 \qquad \Gamma_2 \vdash \eta(v_2) \Rightarrow b_3, \Gamma_3\end{array}}{\Gamma \vdash (\ominus\eta^{\tau})(e) \Rightarrow (b_2 \wedge b_3), \Gamma_3}$$

GUARD

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow^{+} v_1, \Gamma_1\\ \Gamma_1 \vdash \pi(v_1) \Rightarrow b_2, \Gamma_2 \qquad \Gamma_2 \overset{eval}{\vdash} \xi \Rightarrow b_3, \Gamma_3\end{array}}{\Gamma \vdash (\pi \vDash \xi)(e) \Rightarrow (b_2 \wedge b_3), \Gamma_3}$$

TREE-NULLPTR

$$\frac{\Gamma \vdash e \Rightarrow^{+} v, \Gamma_0 \qquad \Gamma_0 \vdash dc\langle\tau\rangle(v) \Rightarrow^{+} \bot, \Gamma_1}{\Gamma \vdash (\text{match}\langle\tau\,[,l]\rangle(\omega_1, ..., \omega_k))(e) \Rightarrow \bot, \Gamma_1}$$

TREE-FALSE

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow^{+} v, \Gamma_0 \qquad \Gamma_0 \vdash dc\langle\tau\rangle(v) \Rightarrow^{+} u^{\tau}, \Gamma_1\\ \Gamma_1 \vdash \Delta_1^{\tau,l}(u) \Rightarrow^{+} v_1, \Gamma_1'\\ \Gamma_1' \overset{flt}{\vdash} \omega_1 \Rightarrow \pi_1 \qquad \Gamma_1' \vdash \pi_1(v_1) \Rightarrow true, \Gamma_2\\ \Gamma_2 \vdash \Delta_2^{\tau,l}(u) \Rightarrow^{+} v_2, \Gamma_2'\\ \Gamma_2' \overset{flt}{\vdash} \omega_2 \Rightarrow \pi_2 \qquad \Gamma_2' \vdash \pi_2(v_2) \Rightarrow true, \Gamma_3\\ \cdots\\ \Gamma_{i-1} \vdash \Delta_{i-1}^{\tau,l}(u) \Rightarrow^{+} v_{i-1}, \Gamma_{i-1}'\\ \Gamma_{i-1}' \overset{flt}{\vdash} \omega_{i-1} \Rightarrow \pi_{i-1} \qquad \Gamma_{i-1}' \vdash \pi_{i-1}(v_{i-1}) \Rightarrow true, \Gamma_i\\ \Gamma_i \vdash \Delta_i^{\tau,l}(u) \Rightarrow^{+} v_i, \Gamma_i'\\ \Gamma_i' \overset{flt}{\vdash} \omega_i \Rightarrow \pi_i \qquad \Gamma_i' \vdash \pi_i(v_i) \Rightarrow false, \Gamma_{i+1}\end{array}}{\Gamma \vdash (\text{match}\langle\tau\,[,l]\rangle(\omega_1, ..., \omega_k))(e) \Rightarrow \bot, \Gamma_{i+1}}$$

TREE-TRUE

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow^{+} v, \Gamma_0 \qquad \Gamma_0 \vdash dc\langle\tau\rangle(v) \Rightarrow^{+} u^{\tau}, \Gamma_1\\ \Gamma_1 \vdash \Delta_1^{\tau,l}(u) \Rightarrow^{+} v_1, \Gamma_1'\\ \Gamma_1' \overset{flt}{\vdash} \omega_1 \Rightarrow \pi_1 \qquad \Gamma_1' \vdash \pi_1(v_1) \Rightarrow true, \Gamma_2\\ \Gamma_2 \vdash \Delta_2^{\tau,l}(u) \Rightarrow^{+} v_2, \Gamma_2'\\ \Gamma_2' \overset{flt}{\vdash} \omega_2 \Rightarrow \pi_2 \qquad \Gamma_2' \vdash \pi_2(v_2) \Rightarrow true, \Gamma_3\\ \cdots\\ \Gamma_k \vdash \Delta_k^{\tau,l}(u) \Rightarrow^{+} v_k, \Gamma_k'\\ \Gamma_k' \overset{flt}{\vdash} \omega_k \Rightarrow \pi_k \qquad \Gamma_k' \vdash \pi_k(v_k) \Rightarrow true, \Gamma_{k+1}\end{array}}{\Gamma \vdash (\text{match}\langle\tau\,[,l]\rangle(\omega_1, ..., \omega_k))(e) \Rightarrow u^{\tau}, \Gamma_{i+1}}$$

Figure 3. Semantics of match-expressions

- Absence of solution that would result in a given value should be indicated through rejection of the pattern.
- Presence of a unique solution should be indicated with acceptance of the pattern and binding of corresponding variables to the solution.

As to the case when multiple solutions are possible, several alternatives can be viable:

- Reject the pattern.
- Accept, binding to either an arbitrary or some normalized solution.
- When set of solutions is guaranteed to be finite – accept, binding solutions to a set variable.
- When set of solutions is guaranteed to be enumerable – accept, binding solutions to a generator capable of enumerating them all.

We believe that depending on application, any of these semantic choices can be valid, which is why we prefer not to make such choice for the user, but rather provide him with means to decide himself. This is why our semantics for matching against generalized n+k pattern depends on a family of user-defined functions

$$\Psi_f^{\tau} : \tau_r \times \tau_1 \times ... \times 1 \times ... \times \tau_k \to bool \times \tau$$

such that $\Psi_f^{\tau}[r](c_1, ..., \bullet, ..., c_k)$ for a given function

$$f : \tau_1, ..., \tau, ..., \tau_k \to \tau_r$$

should return a pair composed of a boolean indicating acceptance of a pattern $f(c_1, ..., x^{\tau}, ..., c_k) = r$ and a solution with respect to x when the match was reported successful. Symbol $\bullet$ indicates a position in the argument list for which a solution is sought, the rest of the arguments are known values. We describe how user can supply the function Ψ_f^{τ} for his own operation f in §6.

The only difference between the three rules defining semantics of n+k patterns is the arity of the root operator and the position in the argument list in which the only non-value pattern was spotted (N+K BINARY LEFT, N+K BINARY RIGHT and N+K UNARY). We use user-defined Ψ_f^{τ} to solve for a given argument of a given operator and then recurse to match corresponding sub-expression. Note that when user did not provide Ψ_f^{τ} for the argument in which solution is sought, the rule is rejected.

Evaluation of a guard pattern first tries to match the left hand side of a guard expression, usually binding a variable in it, and then if the match was successful, lazily evaluates its right hand side to make sure the new value of the bound variable is used. The result of evaluating the right hand side converted to **bool** is reported as the result of matching the entire pattern (GUARD).

Matching of a tree patterns begins with evaluating subject expression and casting it dynamically to target type τ when subject type is not τ. When **dynamic_cast** fails returning nullptr, the pattern is rejected (TREE-NULLPTR). Once a value of a target type has been uncovered, we proceed with matching arguments left-to-right. For each argument we first translate *extended pattern* ω accepted by tree pattern into an *application pattern* π to get rid of the syntactic convenience we allow on arguments of tree patterns. Using the target type and optional layout that was provided by the user, we obtain a value that should be bound in the i^{th} position of decomposition. Again we rely on family of user provided functions $\Delta_i^{\tau,l}(u)$ that take an object instance and returns a value bound in the i^{th} position of its l^{th} layout. Specification of function Δ by the user is discussed in §4.1. Here we would like to point out however that the number of argument patterns passed over to the tree pattern could be smaller than the number of binding positions defined by specific layout. Remaining argument positions are implicitly assumed to be filled with the wildcard pattern.

When we have the value for the i^{th} position of object's decomposition, we match it against the pattern specified in the i^{th} position and if the value is accepted we move on to matching the next argument (TREE-FALSE). Only in case when all the argument patterns have been successfully matched the matching succeeds by returning a pointer to the target type, which in C++ can be used everywhere a boolean expression is expected. Returning pointer instead of just boolean value gives us functionality similar to that of *as patterns* while maintaining the composibility with the rest of the library (TREE-TRUE).

5.2 Semantics of Match Statement

Our second rule set deals with semantics of a *match statement*. The judgements are of the form $\Gamma \vdash s \Rightarrow u, \Gamma'$ on statements, including match statement, and are slightly extended for case clauses $\Gamma \vdash_v C \Rightarrow u, \Gamma'$ with value v of a subject that is passed along from the match statement onto the clauses. We also use a small helper function $TL(t, \tau_s)$ defined on target expression $t \in T$ and the subject's type τ_s:

$$\begin{array}{rcl} TL(\tau, \tau_s) & = & \langle \tau, default\_layout \rangle \\ TL(l, \tau_s) & = & \langle \tau_s, l \rangle \\ TL(view\langle \tau, l \rangle, \tau_s) & = & \langle \tau, l \rangle \end{array}$$

The function essentially disambiguates one of the three kinds of target expressions and returns a combination of a target type and layout used in each case.

MATCH-TRUE

$$\frac{\begin{array}{c} \Gamma \vdash e \Rightarrow^{+} v, \Gamma_1 \qquad v \neq \bot \qquad \Gamma_1 \vdash_v C_1 \Rightarrow false, \Gamma_2 \\ \Gamma_2 \vdash_v C_2 \Rightarrow false, \Gamma_3 \qquad \cdots \qquad \Gamma_{i-1} \vdash_v C_{i-1} \Rightarrow false, \Gamma_i \\ \Gamma_i \vdash_v C_i \Rightarrow true, \Gamma_{i+1} \qquad \Gamma_{i+1} \vdash \vec{s}_i \Rightarrow^{+} u, \Gamma' \end{array}}{\Gamma \vdash Match(e)\,[C_i \vec{s}_i]^{*}_{i=1..n}\, EndMatch \Rightarrow u, \Gamma' \backslash \{x | x \notin \Gamma_i\}}$$

MATCH-FALSE

$$\frac{\begin{array}{c} \Gamma \vdash e \Rightarrow^{+} v, \Gamma_1 \qquad v \neq \bot \\ \Gamma_1 \vdash_v C_1 \Rightarrow false, \Gamma_2 \qquad \Gamma_2 \vdash_v C_2 \Rightarrow false, \Gamma_3 \qquad \cdots \\ \Gamma_{n-1} \vdash_v C_{n-1} \Rightarrow false, \Gamma_n \qquad \Gamma_n \vdash_v C_n \Rightarrow false, \Gamma_{n+1} \end{array}}{\Gamma \vdash Match(e)\,[C_i \vec{s}_i]^{*}_{i=1..n}\, EndMatch \Rightarrow false, \Gamma_{n+1}}$$

QUE

$$\frac{\begin{array}{c} TL(t, \sigma) = \langle \tau, l \rangle \qquad \Gamma \vdash \mathrm{match}\langle \tau, l \rangle(\vec{\omega})(v) \Rightarrow u, \Gamma' \\ \Gamma'' = (u \neq \bot\ ?\ \Gamma'[\mathrm{matched}^{\tau} \to u] : \Gamma') \end{array}}{\Gamma \vdash_{v^{\sigma}} Que(t, \vec{\omega}) \Rightarrow u, \Gamma''}$$

CASE

$$\frac{\begin{array}{c} \Delta^{t}_{i} : \tau \to \tau_i, i = 1..k \\ \Gamma[x_i^{\tau_i} \to \tau_i()]_{i=1..k} \vdash_v Que(t, x_1, ..., x_k)(v) \Rightarrow u, \Gamma' \\ \Gamma'' = (u \neq \bot\ ?\ \Gamma' : \Gamma' \backslash \{x_i | i = 1..k\}) \end{array}}{\Gamma \vdash_v Case(t, x_1, ..., x_k) \Rightarrow u, \Gamma''}$$

OTHERWISE

$$\frac{\Gamma \vdash Case(\tau, \vec{x})(v) \Rightarrow true, \Gamma'}{\Gamma \vdash_{v^{\tau}} Otherwise(\vec{x}) \Rightarrow true, \Gamma'}$$

Evaluation of a match statement begins with evaluation of subject expression, which is not allowed to result in nullptr. This value is passed along to each of the clauses. The clauses are evaluated in their lexical order until the first one that is not rejected. Statements associated with it are evaluated to form the outcome of the match statement. The resulting environment makes sure that local variables introduced by case clauses are not available after the match statement (MATCH-TRUE). When none of the clauses were accepted, which is only possible when default clause was not specified, the resulting environment might still be different from the initial environment because of variables bound in partial matches during evaluation of clauses (MATCH-FALSE).

Evaluation of a Que-clause is equivalent to evaluation of a corresponding match-expression on a tree pattern. Successful match will introduce a variable matched of type $\tau\&$ that is bound to subject properly casted to the target type τ into the local scope of the clause.

Evaluation of *Case*-clauses amounts of evaluation of Que-clauses in the environment extended with variables passed as arguments to the clause. The variables introduced by the *Case*-clause have the static type of values bound in corresponding positions, which ensures that variable patterns will be irrefutable (CASE). In practice, the variables are of reference type so that no unnecessary copying is happening.

Evaluation of default clause cannot fail because there is no **dynamic_cast** involved neither for the subject nor for implicit local variables: for the former the target type is by definition the subject type, while for the latter the type is chosen to be the type of expected values (OTHERWISE).

5.3 Auxiliary Rules

The next rule set deals with evaluation of expression templates referred to from the previous rule sets via $\overset{eval}{\vdash}$. The judgements are of the form $\Gamma \overset{eval}{\vdash} \xi \Rightarrow v, \Gamma'$ that can be interpreted as given an environment Γ, evaluation of an expression template ξ results in value v and environment Γ'. We refer here to an unspecified semantic function $[\![o]\!]$ that represents C++ semantics of operation o as specified by the C++ standard.

VAR

$$\Gamma \overset{eval}{\vdash} \chi \Rightarrow \Gamma(\chi), \Gamma$$

UNARY

$$\frac{\Gamma \overset{eval}{\vdash} \xi \Rightarrow v, \Gamma_1}{\Gamma \overset{eval}{\vdash} \ominus\xi \Rightarrow [\![\ominus]\!]\, v, \Gamma_1}$$

BINARY

$$\frac{\Gamma \overset{eval}{\vdash} \xi_1 \Rightarrow v_1, \Gamma_1 \qquad \Gamma_1 \overset{eval}{\vdash} \xi_2 \Rightarrow v_2, \Gamma_2}{\Gamma \overset{eval}{\vdash} \xi_1 \oplus \xi_2 \Rightarrow v_1\, [\![\oplus]\!]\, v_2, \Gamma_2}$$

BINARY-LEFT

$$\frac{\Gamma \overset{eval}{\vdash} \xi \Rightarrow v, \Gamma_1}{\Gamma \overset{eval}{\vdash} \xi \oplus c \Rightarrow v\, [\![\oplus]\!]\, c, \Gamma_1}$$

BINARY-RIGHT

$$\frac{\Gamma \overset{eval}{\vdash} \xi \Rightarrow v, \Gamma_1}{\Gamma \overset{eval}{\vdash} c \oplus \xi \Rightarrow c\, [\![\oplus]\!]\, v, \Gamma_1}$$

The rules are quite simple so we do not elaborate them in details. The reason we have two separate rules for the case when one of the arguments is constant expression stems from the idiomatic use of expression templates enabling direct use of constants in operations that already involve expression template arguments.

The next set of rules describes transformation of extended patterns into applicative patterns to get rid of syntactic sugar enabled by extended patterns.

FILTER-PATTERN

$$\Gamma \overset{flt}{\vdash} \pi \Rightarrow \pi$$

FILTER-VARIABLE

$$\Gamma \overset{flt}{\vdash} x \Rightarrow \iota(x)$$

FILTER-CONSTANT

$$\Gamma \overset{flt}{\vdash} c \Rightarrow \varsigma(c)$$

6. Generalized n+k Patterns

Intuitively n+k patterns like $f(x, y) = v$ relate a known result of a given function application to its arguments. The case where multiple unknown arguments are matched against a single result should not be immediately discarded as there are known n-ary functions whose inverse is unique. An example of such function is Cantor pairing function that defines bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$. Even when such mappings are not one-to-one, their restriction to a given argument often is. Most generalizations of n+k patterns seem to agree on the following rules:

- Absence of solution that would result in a given value should be indicated through rejection of the pattern.
- Presence of a unique solution should be indicated with acceptance of the pattern and binding of corresponding variables to the solution.

When multiple solutions are possible, returning a set or an enumerator is not usually considered due to differences in types: variable x representing a solution to $f(x) = c$ is intuitively expected to have the same type as the argument of f, so that it can be applied to x. Rejecting the pattern might be a plausible approach in some applications where multiple solutions are treated as ambiguity. It is incapable of distinguishing absence of a solution from ambigous solution however, which is often desired.

Binding to an arbitrary solution in case of multiple ones might be sensitive as to which solution is chosen: some applications might prefer the smallest/largest one, some the smallest positive etc. We believe that depending on application, different semantic choices can be valid, which is why we prefer not to make such choice for the user, but rather provide him with means to decide. In fact we go even further and do not require the values bound by our generalized n+k pattern to be a solution to the corresponding equation. We do this for several reasons:

- Due to numeric errors, truncation and integer overflow we will rarely obtain the exact algebraic solution.
- Curve fitting can be seen as pattern matching in some application domains.
- Sometimes we might be interested in matching against a projection of a value onto some base and the obtained result may not necessarily yield a solution to matching against the original value.

Consider matching an expression $x + 1$ with variable x of type **char** ranging over $[-128, 127]$ against a value -128. Should the value 127 be considered a solution since 127+1 overflows in char resulting in -128? From the mathematical point of view it should not, but a particular application might accept such a solution for the sake of performance. Similarly, matching $3 * y$ against 1.0 with a variable y of type **double** will result in a number that is slightly different from $\frac{1}{3}$. Should such match be accepted the next logical request a user might have is to be able to match an expression of the form n/m for integer variables n and m against a value 3.1415926 and get the closest fraction to it. Matching against such pattern does not have to be imprecise as one can match it against an object of a class representing rational numbers.

Taking the argument of precision even further one may want to be able to do curve fitting with generalized n+k patterns for the sake of expressive syntax. Consider an object that contains sampling of some random variable. A hypotetical match statement might be querying:

```
match (random_variable) {
    case Gaussian(μ,σ^2): ...   case Poisson(λ): ...   case Bernoulli(p): ...
}
```

Fitting error threshold in such scenario can either be global or passed as a parameter into expressions we are matching against: e.g. **case** Gaussian($0.01, \mu, \sigma^2$):. Again the fitting does not have to be imprecise. Consider a library dealing with polynomials of arbitrary degree. Given a general polynomial we might want to be able to check a few special cases for which analytical solutions to some larger question exist:

```
match (polynomial) { case a*X^1 + 1: ...  case 2*X^2 + b*X^1 + c: ... }
```

X in such scenario is not a variable but a placeholder value of a kind that lets us identify the degree, whose coefficient is sought.

A simpler example of this kind is decomposition of a complex number with Euler's notation $a + b * i$ for scalar variables a and b. With variables, such a generalized n+k pattern is irrefutable for all complex numbers, but when a more specific form is queried (e.g. $3 + b * i$) a given complex number may fail to match such a pattern. While matching, we will project such a complex number along its real and imaginary components and will try matching the operands of addition using those projections. Solutions obtained along each projection may not necessarily combine into the final solution.

What all these examples have in common is not necessarily solving the equation that generalized n+k patterns represent, but the fact that we associate certain notations with certain mathematical entities they represent. Parameters of those expressions are typically associated with the parameterns of the underlying mathematical object and we perform decomposition of that object into parts. The structure of the expression tree 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.

Algebraic decomposition to mathematical entities is what views are to algebraic data types. Consider for example an object representing a 2D line. At different parts of the program we might need to decompose that line differently (hypothetical syntax):

```
if (line matches m*X + c) ...                          // slope-intercept
if (line matches a*X + b*Y = c) ...                    // linear equation
if (line matches (Y-y0)*(x1-x0)=(y1-y0)*(X-x0)) ... // two-points forn
```

As before, X and Y are not variables, but some syntactic entities that let us properly decompose parts. Matching against the slope-intercept notation will not be able to decompose a line of the form $y = c$, but otherwise still looks like solving an equation (even though quantified over all X). The other two notations include equality sign in their expression, which makes our argument that we decompose against a known notation (as opposed to solving some equation) stronger.

6.0.1 Solvers

The above class esssentially defines forward semantics of a family of operations. To define backward semantics of it for the use in n+k patterns, the user defines *solvers* by overloading a function

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

The first argument of a function takes an expression template representing an expression we are matched against, while the second argument represents the expected result. The following example defines a generic solver for multiplication by a constant:

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

Note that we overload not only on the structure of the expression, but also on the properties of their result type (or any other type involved). In particular when the type of the result of the subexpression models FIELD concept, we can rely on presence of unique inverse and simply call division without any additional checks. A similar overload for integral multiplication additionally checks that result is divisible by the constant, before generically forwarding the matching to the first argument of multiplication. This last overload combined with a similar solver for addition of integral types is everything the library needs to properly handle the definition of the fib function from §4.

A solver capable of decomposing a complex value using the Euler's notation is very easy to define by fixing the structure of expression:

```
template ⟨LazyExpression E1, LazyExpression E2⟩requires SameType⟨E1::result_type,E2::result_type⟩
bool solve(const expr⟨addition,
                      expr⟨multiplication,E1,value⟨complex⟨E1::result_type⟩⟩⟩,
                      E2
                     ⟩& e,
           const complex⟨E1::result_type⟩& r);
```

7. Views

Support of multiple bindings through layouts in our library effectively enables a facility similar to Wadler's *views*. Reconsider example from §2 that discusses cartesian and polar representations of complex numbers, demonstrating the notion of view. The same example recoded with our SELL looks as following:

```
// Introduce layouts
enum { cartesian = default_layout , polar };

// Define bindings with them
template ⟨typename T⟩ struct bindings⟨std::complex⟨T⟩⟩
  { CM(0,std::real⟨T⟩); CM(1,std::imag⟨T⟩); };
template ⟨typename T⟩ struct bindings⟨std::complex⟨T⟩, polar⟩
  { CM(0,std::abs⟨T⟩);  CM(1,std::arg⟨T⟩); };

// Define views
template ⟨typename T⟩
  using Cartesian = view⟨std::complex⟨T⟩⟩;
template ⟨typename T⟩
  using Polar     = view⟨std::complex⟨T⟩, polar⟩;

  std::complex⟨double⟩ c;
  double a,b,r,f;

  if (match⟨std::complex⟨double⟩⟩(a,b)(c)) // default
  if (match⟨   Cartesian⟨double⟩⟩(a,b)(c)) // same as above
  if (match⟨       Polar⟨double⟩⟩(r,f)(c)) // view
```

The C++ standard effectively enforces the standard library to use cartesian representation[**?** , §26.4-4]. Knowing that, we choose the cartesian layout to be default, with polar being an alternative layout for complex numbers. We then define bindings for each of these layouts as well as introduce template aliases (an analog of typedefs for parameterized classes) for each of the views. Template class view⟨T,l⟩ defined by the library provides a way to bind together a target type with one of its layouts into a single type. This type can be used everywhere in the library where an original target type was expected, while the library will take care of decoding the type and layout from the view and passing them along where needed.

The first two match expressions are the same and incur no runtime overhead since they use default layout of the underlying type. The third match expression will implicitly convert cartesian representation into polar, thus incurring some overhead. This overhead would have been present in code that depends on polar coordinates anyways, since the user would have had to invoke the corresponding functions manually.

8. Match Statement

Our implementation of pattern matching expressions follows the naive way of essentially interpreting them through backtracking. On one hand, this was a consequence of working in a library setting, where code transformations are much harder to achive. On the other hand, from the very beginning we were trying to find an expressive alternative to object decomposition with either nested dynamic casts or visitor design pattern, and thus were not concerned with pattern matching on multiple arguments, where decision tree approach becomes more efficient. Dealing with single argument certainly leaves less choices for optimization, but does not eliminate them as repeated use of constructor-pattern with the same target type but different argument patterns essentially leads to the same inefficiencies. To tackle this issue in a library setting we rely on and give more control to the library user. For example, we fix the order of evaluation, but let guard-patterns be placed directly on the arguments of a constructor-pattern to let the user benefit from the consciesness of expression, while holding a grip on performance. Similarly, we added Alt sub-clauses to Que-clause to syntactically separate fast type switching from slow sequential evaluation of pattern matching expressions. The fall-through behavior of the *Match*-statement allows the user to achieve the same effect directly with Que-clauses, however the performance overhead involved justified the addition of otherwise syntactic sugar.

The interpretation of pattern matching expressions with expression templates follows very closely the composition of expressions described by abstract syntax in §4 as well as their application to subject expression described by evaluation rules in §5.1. This section thus mainly concentrates on efficient implementation of a match statement as well as unification of its syntax to the three encodings of algebraic data types outlined in §3. The discussion will largely focus on devising an efficient *type-switch*, which is then used by our library as a backbone to the general match statement presented in §5.2.

By encoding algebraic data types with classes we alter their semantics in two important ways: we make them *extensible* as new variants can be added by simply deriving from the base class, as well as *hierarchical* as variants can be inherited from other variants and thus form a subtyping relation between themselves [25]. This is not the case with traditional algebraic data types in functional languages, where the set of variants is *closed*, while the variants are *disjoint*. Some functional languages e.g. ML2000 [2] and Moby [**?**] were experimenting with *hierarchical extensible sum types*, which are closer to object-oriented classes then algebraic data types are, but, interestingly, they did not provide pattern matching facilities on them. Working within a multi-paradigm programming language like C++, we will not be looking at algebraic data types in the closed form they are present in functional languages, but rather in an open/extensible form discussed by Zenger [64], Emir [17], Löh [36], Glew [25] and others. We will thus assume an object-oriented setting where new variants can be added later and form subtyping relations between each other including those through multiple inheritance. We will look separately at polymorphic and tagged class encodings as our handling of these two encodings is significantly different. Before we look into these differences in greater details, however, we would like to look at the problem of type switching without specific implementation in mind as well as properties we would like to seek from such an implementation.

8.1 Type Switch

Functional languages use pattern matching to perform case analysis on a given algebraic data type. In this section we will try to generalize this construct to case analysis of hierarchical and extensible data types. Presence of such a construct will allow for external function definitions by detaching a particular case analysis from the hierarchy it is performed on.

Consider a class B and a set of classes D_i directly or indirectly inherited from it. An object is said to be of the *most derived type* D if it was created by explicitly calling a constructor of that type. The inheritance relation on classes induces a subtyping relation on them, which in turn allows objects of a derived class to be used in places where an object of a base class is expected. The type of variable or parameter referencing such an object is called the *static type* of the object. When object is passed by reference or by pointer, we might end up in a situation where the static type of an object is different from its most derived type, with the latter necessarily being a subtype of the former. The most derived class along with all its base classes that are not base classes of the static type are typically referred to as the *dynamic types* of an object. At each program point the compiler knows the static type of an object, but not its dynamic types.

By *type switch* we will call a control structure taking either a pointer or a reference to an object, called *subject*, and capable of uncovering a reference or a pointer to a full object of a type present in the list of case clauses. Similar control structures exist in many programming languages and date back to at least Simula's Inspect statement [11].

Consider an object of (most derived) type D, pointed to by a variable of static type B∗: e.g. B∗ base = **new** D;. A hypothetical type switch statement, not currently supported by C++, can look as following:

```
switch (base)
{
    case $D_1$: $s_1$;
    ...
    case $D_n$: $s_n$;
}
```

and can be given numerous plausible semantics:

- *First-fit* semantics will evaluate the first statement s_i such that D_i is a base class of D
- *Best-fit* semantics will evaluate the statement corresponding to the most derived base class D_i of D if it is unique (subject to ambiguity)
- *The-only-fit* semantics will only evaluate statement s_i if $D_i = D$.
- *All-fit* semantics will evaluate all statements s_i whose guard type D_i is a subtype of D (order of execution has to be defined)
- *Any-fit* semantics might choose non-deterministically one of the statements enabled by all-fit

The list is not exhaustive and depending on a language, any of these semantics or their combination might be a plausible choice. Functional languages, for example, often prefer first-fit, while object-oriented languages would typically be inclined to best-fit semantics. The-only-fit semantics is traditionally seen in procedural languages like C and Pascal to deal with discriminated union types. All-fit and any-fit semantics might be seen in languages based on predicate dispatching [18] or guarded commands [12], where a predicate can be seen as a characteristic function of a type, while logical implication can be seen as subtyping.

8.2 Open and Efficient Type Switching

The fact that algebraic data types in functional languages are closed allows for their efficient implementation. The traditional compilation scheme assigns unique tags to every variant of the algebraic data type and pattern matching is then simply implemented with a jump table over all tags. A number of issues in object-oriented languages makes this extremely efficient approach infeasible:

- Extensibility
- Subtyping
- Multiple inheritance
- Separate compilation
- Dynamic linking

Unlike functional style algebraic data types, classes are *extensible* whereby new variants can be arbitrarily added to the base class in the form of derived classes. Such extension can happen in a different translation unit or a static library (subject to *separate compilation*) or a dynamically linked module (subject to *dynamic linking*). Separate compilation effectively implies that all the derived classes of a given class will only be known at link time, postponing thus any tag-allocation related decisions until then. The Presence of dynamic linking effectively requires the compiler to assume that the exact derived classes will only be known at run time, and not even at start-up time.

The *subtyping* relation that comes along with extensibility through subclassing effectively gives every class multiple types – its own and the types of all its base classes. In such a scenario it is natural to require that type switching can be done not only against the exact dynamic type of an object, but also against any of its base classes (subject to our substitutability requirement). This in itself is not a problem for functional-style tag allocation as long as the set of all derived classes is known, since the compiler can partition tags of all the derived classes according to chosen semantics based on classes mentioned in case clauses. Unfortunately this will not work in the presence of dynamic linking as there might be new derived classes with tags not known at the time of partitioning and thus not mentioned in the generated jump table.

Multiple inheritance complicates things further by making each class potentially belong to numerous unrelated hierarchies. Any tag allocation scheme capable of dealing with multiple inheritance will either have to assure that generated tags satisfy properties of each subhierarchy independently or use different tags for different subhierarchies. Multiple inheritance also introduces such a phenomenon as *cross-casting*, whereby a user may request to cast pointers between unrelated classes, since they can potentially become base classes of a later defined class. From an implementation point of view this means that not only do we have to be able to check that a given object belongs to a given class (type testing), but also be able to find a correct offset to it from a given base class (type casting).

While looking at various schemes for implementing type switching we noted down a few questions that might help evaluate and compare solutions:

1. Can the solution handle base classes in case clauses?
2. Will it handle the presence of base and derived classes in the same match statement?
3. Will it work with derived classes coming from a DLL?
4. Can it cope with multiple inheritance (repeated, virtual)?
5. Can independently developed DLLs that either extend classes involved in type switching or do type switching themselves be loaded together without any integration efforts?
6. Are there any limitations on the number and or shape of class extensions?
7. What is the complexity of performing matching, based on the number of case clauses and the number of possible types?

The number of possible types in the last question refers to the number of subtypes of the static type of the subject, not all the types in the program. Several solutions discussed below depend on the number of case clauses in the match statement, which raises the question of how many such clauses a typical program might have. The C++ pretty-printer for Pivot we implemented using our pattern matching techniques originally had 8 match statements with 5, 7, 8, 10, 15, 17, 30 and 63 case clauses each. While experimenting with probability distributions of various classes to minimize the number of conflicts (see §9.3.2), we had to associate probabilities with classes and implemented it with a match statement over all 160 nodes in the Pivot's class hierarchy. With Pivot having the smallest number of node kinds among the compiler frameworks we had a chance to work with, we expect a similar or larger number of case clauses in other compiler applications.

Instead of starting with an efficient solution and trying to make it open, let us start with an open solution and try to make it efficient. An obvious solution that will pass the above checklist can look like the following:

if $(D_1*$ derived $=$ **dynamic_cast**$\langle D_1* \rangle$(base)) $\{ s_1; \}$ **else**
if $(D_2*$ derived $=$ **dynamic_cast**$\langle D_2* \rangle$(base)) $\{ s_2; \}$ **else**
...
if $(D_n*$ derived $=$ **dynamic_cast**$\langle D_n* \rangle$(base)) $\{ s_n; \}$

Despite the obvious simplicity, its main drawback is performance: a typical implementation of **dynamic_cast** might take time proportional to the distance between base and derived classes in the inheritance tree [**?**]. What is worse, is that the time to uncover the type in the i^{th} case clause is proportional to i, while failure to match will always take the longest. This linear increase can be seen in the Figure 4, where the above cascading-if was applied to a flat hierarchy encoding an algebraic data type with 100 variants. The same type-switching functionality implemented with the visitor design pattern took only 28 cycles regardless of the case.[4] This is more than 3 times faster than the 93 cycles it took to uncover even the first case with **dynamic_cast**, while it took 22760 cycles to uncover the last.

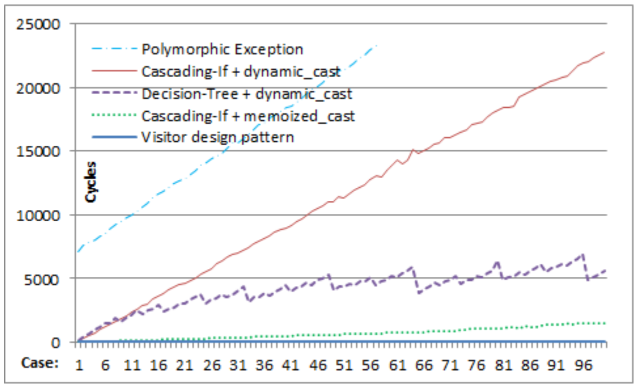

Figure 4. Type switching based on naïve techniques

When the class hierarchy is not flat and has several levels, the above cascading-if can be replaced with a decision tree that tests base classes first and thus eliminates many of the derived classes from consideration. This approach is used by Emir to deal with type patterns in Scala [17, §4.2]. The intent is to replace a sequence of independent dynamic casts between classes that are far from each other in the hierarchy with nested dynamic casts between classes that are close to each other. Another advantage is the possibility to fail early: if the type of the subject does not match any of the clauses, we will not have to try all the cases. A flat hierarchy, which will likely be formed by the leaves in even a multi-level hierarchy, will not be able to benefit from this optimization and will effectively degrade to the above cascading-if. Nevertheless, when applicable, the optimization can be very useful and its benefits can be seen in Figure 4 under "Decision-Tree + dynamic_cast". The class hierarchy for this timing experiment formed a perfect binary tree with classes number 2*N and 2*N+1 derived from a class with number N. The structure of the hierarchy also explains the repetitive pattern of timings.

The above solution either in a form of cascading-if or as a decision tree can be significantly improved by lowering the cost of a single **dynamic_cast**. We devised an asymptotically constant version of this operator that we call memoized_cast in §13. As can be seen from the graph titled "Cascading-If + memoized_cast", it speeds up the above cascading-if solution by a factor of 18 on average, as well as outperforms the decision-tree based solution with dynamic_cast for a number of case clauses way beyond those that can happen in a reasonable program. We leave the discussion of the technique until §13, while we keep it in the chart to give perspective on an even faster solution to dynamic casting. The slowest implementation in the chart based on exception handling facilities of C++ is discussed in §11.

The approach of Gibbs and Stroustrup [22] employs divisibility of numbers to obtain a tag allocation scheme capable of performing type testing in constant time. Extended with a mechanism for storing offsets required for this-pointer adjustments, the technique can be used for extremely fast dynamic casting on quite large class hierarchies. The idea is to allocate tags for each class in such a way that tag of a class D is divisible by a tag of a class B if and only if class D is derived from class B. For comparison purposes we hand crafted this technique on the above flat and binary-tree hierarchies and then redid the timing experiments from Figure 4 using the fast dynamic cast. The results are presented in Figure 5. For reference purposes we retained "Visitor Design Pattern" and "Cascading-If + memoized_cast" timings from Figure 4 unchanged. Note that the Y-axis has been scaled-up 140 times, which is why the slope of "Cascading-If + memoized_cast" timings is so much steeper.

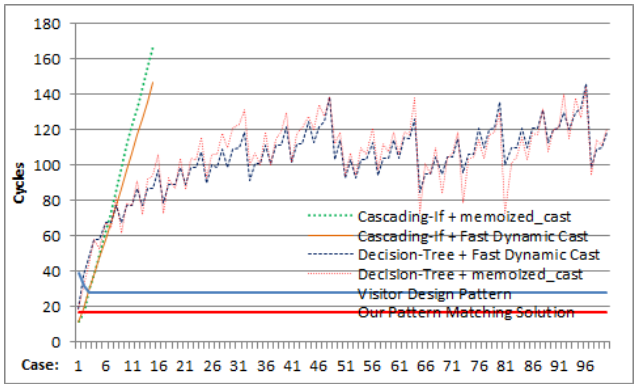

Figure 5. Type switching based on the fast dynamic cast of Gibbs and Stroustrup [22]

As can be seen from the figure the use of our memoized_cast implementation can get close in terms of performance to the fast dynamic cast, especially when combined with decision trees. An important difference that cannot be seen from the chart, however, is that the performance of memoized_cast is asymptotic, while the performance of fast dynamic cast is guaranteed. This happens because the implementation of memoized_cast will incur an overhead of a regular dynamic_cast call on every first call with a given most derived type. Once that class is memoized, the performance will remain as shown. Averaged over all calls with a given type we can only claim we are asymptotically as good as fast dynamic cast.

[4] Each case i was timed multiple times, thus turning the experiment into a repetitive benchmark described in §14. In a more realistic setting, represented by random and sequential benchmarks, the cost of double dispatch was varying between 52 and 55 cycles.

Unfortunately fast dynamic casting is not truly open to fully satisfy our checklist. The structure of tags required by the scheme limits the number of classes it can handle. A 32-bit integer is estimated to be able to represent 7 levels of a class hierarchy that forms a binary tree (255 classes), 6 levels of a similar ternary tree hierarchy (1093 classes) or just one level of a hierarchy with 9 base classes – multiple inheritance is the worst case scenario of the scheme that quickly drains its allocation possibilities. Besides, similarly to other tag allocation schemes, presence of class extensions in *Dynamically Linked Libraries* (DLLs) will likely require an integration effort to make sure different DLLs are not reusing prime numbers in a way that might result in an incorrect dynamic cast.

A number of other constant-time techniques for class-membership testing is surveyed by Gil and Zibin [23, §4]. They are intended for type testing, and thus will have to be combined with decision trees for type switching, resulting in similar to fast dynamic cast performance. They too assume access to the entire class hierarchy at compile time and thus are not open.

In view of the predictably-constant dispatching overhead of the visitor design pattern, it is clear that any open solution that will have a non-constant dispatching overhead will have a poor chance of being adopted. Multi-way switch on sequentially allocated tags [53] was one of the few techniques that could achieve constant overhead, and thus compete with and even outperform visitors. Unfortunately the scheme has problems of its own that make it unsuitable for truly open type-switching and here is why.

The simple scheme of assigning a unique tag per variant (instantiatable class here) will not pass our first question because the tags of base and derived classes will have to be different if the base class can be instantiated on its own. In other words we will not be able to land on a case label of a base class, while having a derived tag only. The already mentioned partitioning of tags of derived classes based on the classes in case clauses also will not help as it assumes knowledge of all the classes and thus fails extensibility through DLLs.

In practical implementations hand crafted for a specific class hierarchy, tags often are not chosen arbitrarily, but to reflect the subtyping relation of the underlying hierarchy. Switching on base classes in such a setting will typically involve a call to some function f that converts derived class' tag into a base class' tag. An example of such a scheme would be having a certain bit in the tag set for all the classes derived from a given base class. Unfortunately this solution creates more problems than it solves.

First of all the solution will not be able to recognize an exceptional case where most of the derived classes should be handled as a base class, while a few should be handled specifically. Applying the function f puts several different types into an equivalence class with their base type, making them indistinguishable from each other.

Secondly, the assumed structure of tags is likely to make the set of tags sparse, effectively forcing the compiler to use a decision tree instead of a jump table to implement the switch. Even though conditional jump is reported to be faster than indirect jump on many computer architectures [21, §4], this did not seem to be the case in our experiments. Splitting of a jump table into two with a condition, that was sometimes happening because of our case label allocation scheme, was resulting in a noticeable degradation of performance in comparison to a single jump table.

Besides, as was seen in the scheme of Gibbs and Stroustrup, the assumed structure of tags can also significantly decrease the number of classes a given allocation scheme can handle. It is also interesting to note that even though their scheme can be easily adopted for type switching with decision trees, it is not easily adoptable for type switching with jump tables: in order to obtain tags of base classes we will have to decompose the derived tag into primes and then find all the dividers of the tag present in case clauses.

To summarize, truly open and efficient type switching is a non-trivial problem. The approaches we found in the literature were either open or efficient, but not both. Efficient implementation was typically achieved by sealing the class hierarchy and using a jump table on sequential tags. Open implementations were resorting to type testing and decision trees, which was not efficient. We are unaware of any efficient tag allocation scheme that can be used in a truly open scenario.

9. Solution for Polymorphic Classes

Our handling of type switches for polymorphic and tagged encodings differs with each having its pros and cons described in details in §14.1. In this section we will concentrate on the truly open type switch for polymorphic encoding. The type switch for tagged encoding (§10) is simpler and more efficient, however, making it open will eradicate its performance advantages. The difference in performance is the price we pay for keeping the solution open. The core of the proposal relies on two key aspects of C++ implementations:

1. a constant-time access to the virtual table pointer embedded in an object of dynamic class type;
2. injectivity of the relation between an object's inheritance path and the virtual table pointer extracted from that object.

9.1 Virtual Table Pointers

Before we discuss our solution we would like to talk about certain properties of the C++ run-time system that we rely on. In particular, we show that under certain conditions the compiler cannot share the same virtual tables between different classes or subobjects of the same class. This allows us to use virtual table pointers to *uniquely* identify the subobjects within the most derived class.

Strictly speaking, the C++ standard [27] does not require implementations to use any specific technique (e.g. virtual tables) to implement virtual functions, however interoperability requirements have forced many compiler vendors to design a set of rules called Common Vendor Application Binary Interface (the C++ ABI) [7]. Most C++ compilers today follow these rules, with the notable exception of Microsoft Visual C++. The technique presented here will work with any C++ compiler that follows the C++ ABI. Microsoft's own ABI is not publically available and thus we cannot formally verify that it satisfies our requirements. Nevertheless, we did run numerous experiments with various class hierarchies and have sufficient confidence that our approach can be used in Visual C++. This is why we include experimental results for this compiler as well.

Besides single inheritance, which is supported by most object-oriented languages, C++ supports multiple-inheritance of two kinds: repeated and virtual (shared). *Repeated inheritance* creates multiple independent subobjects of the same type within the most derived type. *Virtual inheritance* creates only one shared subobject, regardless of the inheritance paths. Because of this peculiarity of the C++ type system it is not sufficient to talk only about the static and dynamic types of an object – one has to talk about a *subobject* of a certain static type accessible through a given inheritance path within a dynamic type.
Note that the above picture portrais subobject relatedion, not the inheritance.

The notion of subobject has been formalized before [47, 49, 63]. We follow here the presentation of Ramamanandro et al [47].

A base class subobject of a given complete object is represented by a pair $\sigma = \langle h, l \rangle$ with $h \in \{\mathrm{Repeated}, \mathrm{Shared}\}$ representing the kind of inheritance (single inheritance is $\mathrm{Repeated}$ with one base class) and l representing the path in a non-virtual inheritance graph.

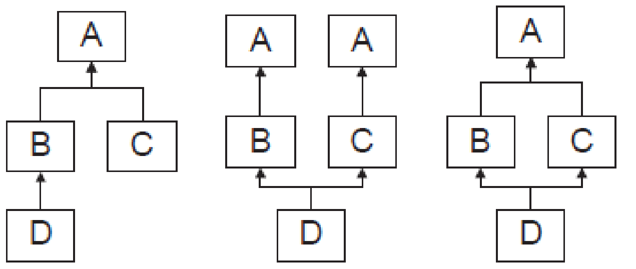

Figure 6. Single inheritance, repeated multiple inheritance and virtual multiple inheritance

A predicate $C \prec \sigma \succ A$ they introduce means that σ designates a subobject of static type A within the most derived object of type C.

A class that declares or inherits a virtual function is called a *polymorphic class* [27, §10.3]. The C++ ABI in turn defines *dynamic class* to be a class requiring a virtual table pointer (because it or its bases have one or more virtual member functions or virtual base classes). A polymorphic class is thus a dynamic class by definition.

A *virtual table pointer* (vtbl-pointer) is a member of object's layout pointing to a virtual table. A *virtual table* is a table of information used to dispatch virtual functions, access virtual base class subobjects, and to access information for *RunTime Type Identification* (RTTI). Because of repeated inheritance, an object of given type may have several vtbl-pointers in it. Each such pointer corresponds to one of the polymorphic base classes. Given an object a of static type A that has k vtbl-pointers in it, we will use the same notation we use for regular fields to refer them: $a.vtbl_i$.

A *primary base class* for a dynamic class is the unique base class (if any) with which it shares the virtual table pointer at offset 0. The data layout procedure for non-POD types described in §2.4 of the C++ ABI [7] requires dynamic classes either to allocate vtable pointer at offset 0 or share the virtual table pointer from its primary base class, which is by definition at offset 0. For our purpose this means that we can rely on a virtual table pointer always being present at offset 0 for all dynamic classes, and thus for all polymorphic classes.

Lemma 1. *In an object layout that adheres to the C++ ABI, a polymorphic class always has a virtual table pointer at offset 0.*

Knowing how to extract a vtbl-pointer as well as that all the objects of the same most derived type share the same vtbl-pointers, the idea is to use their values to uniquely identify the type and subobject within it. Unfortunately nothing in the C++ ABI states these pointers should be unique. A popular optimization technique lets the compiler share the virtual table of a derived class with its primary base class as long as the derived class that does not override any virtual methods. Use of such optimization will violate the uniqueness of vtbl-pointers; however, we show below that in the presense of RTTI, a C++ ABI-compliant implementation is guaranteed to have different values of vtbl-pointers in different subobjects.

The exact content of the virtual table is not important for our discussion, but we would like to point out a few fields in it. The following definitions are copied verbatim from the C++ ABI [7, §2.5.2]:

- The *typeinfo pointer* points to the typeinfo object used for RTTI. It is always present.
- The *offset to top* holds the displacement to the top of the object from the location within the object of the virtual table pointer that addresses this virtual table, as a ptrdiff_t. It is always present.
- *Virtual Base (vbase) offsets* are used to access the virtual bases of an object. Such an entry is added to the derived class object address (i.e. the address of its virtual table pointer) to get the address of a virtual base class subobject. Such an entry is required for each virtual base class.

Given a virtual table pointer vtbl, we will refer to these fields as rtti(vtbl), off2top(vtbl) and vbase(vtbl) respectively. We will also assume presence of a function $offset(\sigma)$ that defines the offset of the base class identified by the end of the path σ within a class identified by its first element.

Theorem 1. *In an object layout that adheres to the C++ ABI with present runtime type information, the equality of virtual table pointers of two objects of the same static type implies that they both belong to subobjects with the same inheritance path in the same most-derived type.*

$$\forall a_1, a_2 : A \mid a_1 \in C_1 \prec \sigma_1 \succ A \wedge a_2 \in C_2 \prec \sigma_2 \succ A$$
$$a_1.vtbl_i = a_2.vtbl_i \Rightarrow C_1 = C_2 \wedge \sigma_1 = \sigma_2$$

Proof. Let us assume first $a_1.vtbl_i = a_2.vtbl_i$ but $C_1 \neq C_2$. In this case we have rtti$(a_1.vtbl_i)$ =rtti$(a_2.vtbl_i)$. By definition rtti$(a_1.vtbl_i) = C_1$ while rtti$(a_2.vtbl_i) = C_2$, which contradicts that $C_1 \neq C_2$. Thus $C_1 = C_2 = C$.

Let us assume now that $a_1.vtbl_i = a_2.vtbl_i$ but $\sigma_1 \neq \sigma_2$. Let $\sigma_i = \langle h_i, l_i \rangle, i = 1, 2$

If $h_1 \neq h_2$ then one of them refers to a virtual base while the other to a repeated one. Assuming h_1 refers to a virtual path, vbase$(a_1.vtbl_i)$ has to be defined inside the vtable according to the ABI, while vbase$(a_2.vtbl_i)$ – should not. This would contradict again that both $vtbl_i$ refer to the same virtual table.

We thus have $h_1 = h_2 = h$. If $h = \text{Shared}$ then there is only one path to such A in C, which would contradict $\sigma_1 \neq \sigma_2$. If $h = \text{Repeated}$ then we must have that $l_1 \neq l_2$. In this case let k be the first position in which they differ: $l_1^j = l_2^j \forall j < k \wedge l_1^k \neq l_2^k$. Since our class A is a base class for classes l_1^k and l_2^k, both of which are in turn base classes of C, the object identity requirement of C++ requires that the relevant subobjects of type A have different offsets within class C: $offset(\sigma_1) \neq offset(\sigma_2)$ However $offset(\sigma_1)$ =off2top$(a_1.vtbl_i)$ =off2top$(a_2.vtbl_i)$ = $offset(\sigma_2)$ since $a_1.vtbl_i = a_2.vtbl_i$, which contradicts that the offsets are different. □

Conjecture in the other direction is not true in general as there may be duplicate virtual tables for the same type present at runtime. This happens in many C++ implementations in the presence of DLLs as the same class compiled into executable and into a DLL it loads may have identical virtual tables inside the executable's and DLL's binaries.

Note also that we require both static types to be the same. Dropping this requirement and saying that equality of vtbl-pointers also implies equality of the static types is not true in general because a derived class will share the vtbl-pointer with its primary base class (see Lemma 1). The theorem can be reformulated, however, stating that one static type will necessarily have to be a subtype of the other. The current formulation is sufficient for our purposes, while reformulation would have required more elaborate discussion of the algebra of subobjects [47], which we touch only briefly.

Corollary 1. *Results of* **dynamic_cast** *can be reapplied to a different instance from within the same subobject.*

$\forall A, B \forall a_1, a_2 : A \mid a_1.vtbl_i = a_2.vtbl_i \Rightarrow$
dynamic_cast$\langle B\rangle(a_1).vtbl_j =$**dynamic_cast**$\langle B\rangle(a_2).vtbl_j \vee$
dynamic_cast$\langle B\rangle(a_1)$ *throws* $\wedge$ **dynamic_cast**$\langle B\rangle(a_2)$ *throws.*

During construction and deconstruction of an object, the value of a given vtbl-pointer may change. In particular, that value will reflect the dynamic type of the object to be the type of the fully constructed part only. However, this does not affect our reasoning, as during such transition we also treat the object to have the type of its fully constructed base only. Such interpretation is in line with the C++ semantics for virtual function calls and the use of RTTI during construction and destruction of an object. Once the complete object is fully constructed, the value of the vtbl-pointer will remain the same for the lifetime of the object.

9.2 Memoization Device

Let us look at a slightly more general problem than type switching. Consider a generalization of the switch statement that takes predicates on a subject as its clauses and executes the first statement s_i whose predicate is enabled:

```
switch (x)
{
    case $P_1$(x): $s_1$;
    ...
    case $P_n$(x): $s_n$;
}
```

Assuming that predicates depend only on x and nothing else as well as that they do not involve any side effects, we can be sure that the next time we come to such a switch with the same value, the same predicate will be enabled first. Thus, we would like to avoid evaluating predicates and jump straight to the statement it guards. In a way we would like the switch to memoize which case is enabled for a given value of x.

The idea is to generate a simple cascading-if statement interleaved with jump targets and instructions that associate the original value with enabled target. The code before the statement looks up whether the association for a given value has already been established, and, if so, jumps directly to the target; otherwise the sequential execution of the cascading-if is started. To ensure that the actual code associated with the predicates remains unaware of this optimization, the code preceeding it after the target must re-establish any invariant guaranteed by sequential execution (§9.3).

The above code can easily be produced in a compiler setting, but producing it in a library setting is a challenge. Inspired by Duff's Device [56], we devised a construct that we call *Memoization Device* that does just that in standard C++:

```
typedef decltype(x) T;
static std::unordered_map⟨T,int⟩ jump_target_map;

switch (int& target = jump_target_map[x])
{
default: // entered when we have not seen x yet
    if ($P_1$(x)) { target = 1; case 1: $s_1$;} else
    if ($P_2$(x)) { target = 2; case 2: $s_2$;} else
  ...
    if ($P_n$(x)) { target = $n$; case $n$: $s_n$;} else
                  target = $n + 1$;
case $n + 1$: // none of the predicates is true on x
}
```

The static jump_target_map hash table will be allocated upon first entry to the function. The map is initially empty and according to its logic, request for a key x not yet in the map will result in allocation of a new entry with its associated data default initialized (to 0 for int). Since there is no case label 0 in the switch, the default case will be taken, which, in turn, will initiate sequential execution of the interleaved cascading-if statement. Assignments to target effectively establish association between value x and corresponding predicate, since target is just a reference to jump_target_map[x]. The last assignment records absence of enabled predicates for the value.

The sequential execution of the cascading-if statement will keep checking predicates $P_j(x)$ until the first predicate $P_i(x)$ that returns true. By assigning i to target we will effectively associate i with x since target is just a reference to jump_target_map[x]. This association will make sure that the next time we are called with the value x we will jump directly to the label i. When none of the predicates returns true, we will record it by associating x with $N+1$, so that the next time we can jump directly to the end of the switch on x.

The above construct effectively gives the entire statement first-fit semantics. In order to evaluate all the statements whose predicates are true, and thus give the construct all-fit semantics, we might want to be able to preserve the fall-through behavior of the switch. In this case we can still skip the initial predicates returning false and start from the first successful one. This can be easily achieved by removing all else statements and making if-statements independent as well as wrapping all assignments to target with a condition, to make sure only the first successful predicate executes it:

```
    if ($P_i$(x)) { if (target ==0) target = $i$; case $i$: $s_i$;}
```

Note that the protocol that has to be maintained by this structure does not depend on the actual values of case labels. We only require them to be different and include a predefined default value. The default clause can be replaced with a case clause for the predefined value, however keeping the default clause results in a faster code. A more important performance consideration is to keep the values close to each other. Not following this rule might result in a compiler choosing a decision tree over a jump table implementation of the switch, which in our experience significantly degrades the performance.

The first-fit semantics is not an inherent property of the memoization device however. Assuming that the conditions are either mutually exclusive or imply one another, we can build a decision-tree-based memoization device that will effectively have *most-specific* semantics – an analog of best-fit semantics in predicate dispatching [18].

Imagine that the predicates with the numbers $2i$ and $2i+1$ are mutually exclusive and each imply the value of the predicate with number i i.e. $\forall x \in \mathsf{Domain}(P)$

$$P_{2i+1}(x) \to P_i(x) \wedge P_{2i}(x) \to P_i(x) \wedge \neg(P_{2i+1}(x) \wedge P_{2i}(x))$$

The following decision-tree based memoization device will execute the statement s_i associated with the *most-specific* predicate P_i (i.e. the predicate that implies all other predicates true on x) that evaluates to true or will skip the entire statement if none of the predicates is true on x.

```
switch (int& target = jump_target_map[x])
{
default:
    if ($P_1$(x)) {
        if ($P_2$(x)) {
            if ($P_4$(x)) { target = 4; case 4: $s_4$;} else
            if ($P_5$(x)) { target = 5; case 5: $s_5$;}
            target = 2; case 2: $s_2$;
        } else
        if ($P_3$(x)) {
            if ($P_6$(x)) { target = 6; case 6: $s_6$;} else
            if ($P_7$(x)) { target = 7; case 7: $s_7$;}
            target = 3; case 3: $s_3$;
        }
        target = 1; case 1: $s_1$;
    } else {
        target = 0; case 0: ;
    }
}
```

An example of predicates that satisfy this condition are class membership tests where the truth of a predicate that tests membership in a derived class implies the truth of a predicate that tests membership in its base class. Our library solution prefers the simpler cascading-if approach only because the necessary structure of the code can be laid out directly with macros. A compiler solution will use the decision-tree approach whenever possible to lower the cost of the first match from linear in case's number to logarithmic as seen in Figure4.

When the predicates do not satisfy the implication or mutual exclusion properties mentioned above, a compiler of a language based on predicate dispatching would typically issue an ambiguity error. Some languages might choose to resolve it according to lexical or some other ordering. In any case, the presence of ambiguities or their resolution has nothing to do with memoization device itself. The latter only helps optimize the execution once a particular choice of semantics has been made and code implementing it has been laid out.

The main advantage of the memoization device is that it can be built around almost any code, providing that we can re-establish the invariants, guaranteed by sequential execution. Its main disadvantage is the size of the hash table that grows proportionally to the number of different values seen. Fortunately, the values can often be grouped into equivalence classes, such that values in the same class do not change the predicate. The map can then associate the equivalence class of a value with a target instead of associating the value with it. The next subsection does exactly that for polymorphic objects.

9.3 Vtable Pointer Memoization

The memoization device can almost immediately be used for multi-way type testing by using **dynamic_cast**$\langle D_i \rangle$ as a predicate P_i. This cannot be considered a type switching solution, however, as one would expect to also have a reference to the uncovered type. Using a **static_cast**$\langle D_i \rangle$ upon successful type test would have been a solution if we did not have multiple inheritance. It certainly can be used as such in languages with only single inheritance. For the fully functional C++ solution, we combine the memoization device with the properties of virtual table pointers into a *Vtable Pointer Memoization* technique.

We saw that vtbl-pointers uniquely determine the subobject within an object (Theorem 1), while the result of a **dynamic_cast** can be reapplied from the same subobject (Corollary 1). The idea is thus to group all the objects according to the value of their vtbl-pointer and associate both target and the required offset with it through memoization device:

```
typedef std::pair⟨ptrdiff_t,size_t⟩ type_switch_info;
static std::unordered_map⟨intptr_t, type_switch_info⟩ jump_target_map;
intptr_t vtbl = *reinterpret_cast⟨const intptr_t*⟩(p);
type_switch_info& info = jump_target_map[vtbl];
const void* tptr;
switch (info.second) ...
```

We use the virtual table pointer extracted from a polymorphic object pointed to by p as a key for association. The value stored along the key in association now keeps both: the target for the switch as well as a memoized offset for dynamic cast.

The code for the i^{th} case now evaluates the required offset on the first entry and associates it along the target with the vtbl-pointer of the subject. The call to adjust_ptr$\langle D_i \rangle$ re-establishes the invariant that matched is a properly-casted reference to type D_i of the subject p.

```
    if (tptr = dynamic_cast⟨const $D_i$*⟩(p)) {
        if (info.second ==0) { // supports fall−through
            info.first = intptr_t(tptr)−intptr_t(p); // offset
            info.second = $i$; // jump target
        }
case $i$: // $i$ is a constant here − clause's position in switch
        auto matched = adjust_ptr⟨$D_i$⟩(p,info.first);
        $s_i$;
    }
```

The main condition remains the same. We keep checking for the first initialization because we allow fall-through semantics here, letting the user break from the switch when needed. Upon first entry we compute the offset that the dynamic cast performed and save it together with target associated to the virtual table pointer. On the next iteration we will jump directly to the case label and restore the invariant of matched being a properly-casted reference to the derived object.

The use of dynamic cast makes a huge difference in comparison to the use of static cast we dismissed above. First of all the C++ type system is much more restrictive about the static cast and many cases where it is not allowed can still be handled by dynamic cast. Examples of these include downcasting from an ambiguous base class or cross-casting between unrelated base classes.

An important benefit we get from this optimization is that we do not store the actual values (pointers to objects) in the hash table anymore, but group them into equivalence classes based on their virtual table pointers. The number of such pointers in a program is always bound by $O(|A|)$, where A represents the static type of an object, while $|A|$ represents the number of classes directly or indirectly derived from A. The linear coefficient hidden in big-o notation reflects possibly multiple vtbl-pointers in derived classes due to the use of multiple inheritance.

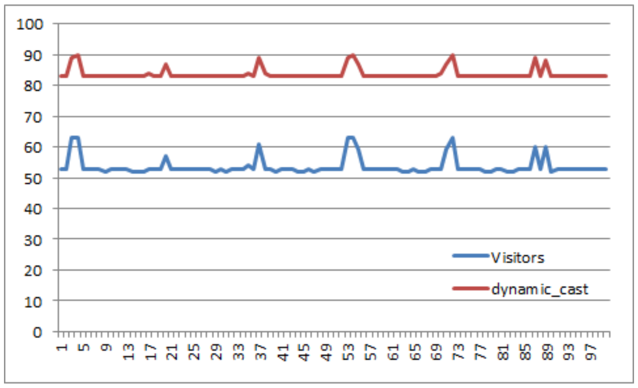

Figure 7. Time to uncover i^th case. X-axis - case i; Y-axis - cycles per iteration

The most important benefit of this optimization, however, is the constant time on average used to dispatch each of the case clauses, regardless of their position in the type switch. The net effect of this optimization can be seen in Figure 7. We can see that the time does not increase with the position of the case we are handling. The spikes represent activities on computer during measurement and are present in both measurements. The constant time on average comes from the average complexity of accessing an element in an unordered_map, while its worst complexity can be proportional to the size of the map. We show in the next section, however, that most of the time we will be bypassing traditional access to elements of the map, because, as-is, the type switch is still about 50% slower than the visitor design pattern.

Note that we can apply the reasoning of §9.2 and change the first-fit semantics of the resulting match statement into a best-fit semantics simply by changing the underlying cascading-if structure with decision tree. A compiler implementation of a type switch based on Vtable Pointer Memoization will certainly take advantage of this optimization to cut down the cost of the first run on a given vtbl-pointer, when the actual memoization happens.

9.3.1 Structure of Virtual Table Pointers

Virtual table pointers are not entirely random addresses in memory and have certain structure when we look at groups of those that are associated with classes related by inheritance. Let us first look at some vtbl pointers that were present in some of our tests. The 32-bit pointers are shown in binary form (lower bits on the right) and are sorted in ascending order:

```
00000001001111100000011001001000
00000001001111100000011001011100
00000001001111100000011001110000
  ...
00000001001111100000011111011000
00000001001111100000011111101100
```

Virtual table pointers are not constant values and are not even guaranteed to be the same between different runs of the same application. Techniques like *address space layout randomization* or simple *rebasing* of the entire module are likely to change these values. The relative distance between them is likely to remain the same though as long as they come from the same module.

Comparing all the vtbl pointers that are coming through a given match statement we can trace ar run time the set of bits in which they do and do not differ. For the above example it may look as `00000001001111100000X11XXXXXXX00` where positions marked with X represent bits that are different in some vtbl pointers.

When a DLL is loaded it may have its own copy of vtables for classes also used in other modules as well as vtables for classes it introduces. Comparing similarly all vtbl pointers coming only from this DLL we can get a different pattern `01110011100000010111XXXXXXXXX000` and when compared over all the loaded modules the pattern will likely becomes something like `0XXX00X1X0XXXXXX0XXXXXXXXXXXXX00`.

The common bits on the right come from the virtual table size and alignment requirements, and, depending on compiler, configuration, and class hierarchy could easily vary from 2 to 6 bits. Because the vtbl-pointer under the C++ ABI points into an array of function pointers, the alignment requirement of 4 bytes for those pointers on a 32-bit architecture is what makes at least the last 2 bits to be 0. For our purpose the exact number of bits on the right is not important as we evaluate this number at run time based on vtbl-pointers seen so far. Here we only would like to point out that there would be some number of common bits on the right.

Another observation we made during our experiments with the vtbl-pointers of various existing applications was that the values of the pointers where changing more frequently in the lower bits than in the higher ones. We believe that this was happening because programmers tend to group multiple derived classes in the same translation unit so the compiler was emitting virtual tables for them close to each other as well.

Note that derived classes that do not introduce their own virtual functions (even if they override some existing ones) are likely to have virtual tables of the same size as their base class. Even when they do add new virtual functions, the size of their virtual tables can only increase relative to their base classes. This is why the difference between many consecutive vtbl-pointers that came through a given match statement was usually constant or very slightly different.

The changes in higher bits were typically due to separate compilation and especially due to dynamically loaded modules. When a DLL is loaded, it may have its own copies of vtables for classes that are also used in other modules, in addition to vtables for classes it introduces. Comparing all vtbl-pointers coming only from that DLL we can get a different pattern `01110011100000010111XXXXXXXXX000` and when compared over all the loaded modules the pattern will likely become something like `0XXX00X1X0XXXXXX0XXXXXXXXXXXXX00`. Overall they were not changing the general tendency we saw: smaller bits were changing more frequently than larger ones, with the exception of the lowest common bits, of course.

These observations made virtual table pointers of classes related by inheritance ideally suitable for indexing – the values obtained by throwing away the common bits on the right were compactly distributed in small disjoint ranges. We use those values to address a cache built on top of the hash table in order to eliminate a hash table lookup in most of the cases. The important guarantee about the validity of the cached hash table references comes from the C++0x standard, which states that "insert and emplace members shall not affect the validity of references to container elements" [27, §23.2.5(13)].

Depending on the number of actual collisions that happen in the cache, our vtable pointer memoization technique can come close to, and even outperform, the visitor design pattern. The numbers are, of course, averaged over many runs as the first run on every vtbl-pointer will take an amount of time as shown in Figure4. We did however test our technique on real code and can confirm that it does perform well in the real-world use cases.

The information about jump targets and necessary offsets is just an example of information we might want to be able to associate with, and access via, virtual table pointers. Our implementation of memoized_cast from §13 effectively reuses this general data structure with a different type of element values. We thus created a generic reusable class vtblmap⟨T⟩ that maps vtbl-pointers to elements of type T. We will refer to the combined cache and hash-table data structure, extended with the logic for minimizing conflicts presented below, as a *vtblmap* data structure.

9.3.2 Minimization of Conflicts

The small number of cycles that the visitor design pattern needs to uncover a type does not let us put too sophisticated cache indexing mechanisms into the critical path of execution. This is why we limit our indexing function to shifts and masking operations as well as choose the size of the cache to be a power of 2.

Throughout this section by *collision* we will call a run-time condition in which the cache entry of an incoming vtbl pointer is occupied by another vtbl-pointer. Collision requires vtblmap to fetch the data associated with the new vtbl-pointer from a slower hash-table and, under certain conditions, reconfigure cache for better performance. By *conflict* we will call a different run-time condition under which given cache configuration maps two or more vtbl pointers to the same cache location. Presence of conflict does not necessarily imply presence of collisions, but collisions can only happen when there is a conflict. In the rest of this section we devise a mechanism that tries to minimize the amount of conflicts in a hope that it will also decrease the amount of actual collisions.

Given n vtbl-pointers we can always find a cache size that will render no conflicts between them. The necessary size of such a cache, however, can be too big to justify the use of memory. This is why, in our current implementation, we always consider only 2 different cache sizes: 2^k and 2^{k+1} where $2^{k-1} < n \le 2^k$. This guarantees that the cache size is never more than 4 times bigger than the minimum required cache size.

During our experiments, we noticed that often the change in the smallest different bit happens only in a few vtbl-pointers, which was effectively cutting the available cache space in half. To overcome this problem, we let the number of bits by which we shift the vtbl-pointer vary further and compute it in a way that minimizes the number of conflicts.

To avoid doing any computations in the critical path, vtblmap only recomputes the optimal shift and the size of the cache when an actual collision happens. In order to avoid constant recomputations when conflicts are unavoidable, we add an additional restriction of only reconfiguring the optimal parameters if the number of vtbl-pointers in the vtblmap has increased since the last recomputation. Since the number of vtbl-pointers is of the order $O(|A|)$, where A is the static type of all vtbl-pointers coming through a vtblmap, the restriction assures that reconfigurations will not happen infinitely often.

To minimize the number of recomputations even further, our library communicates to the vtblmap, through its constructor, the number of case clauses in the underlying match statement. We use this number as an estimate of the expected size of the vtblmap and pre-allocate the cache according to this estimated number. The cache is still allowed to grow based on the actual number of vtbl-pointers that comes through a vtblmap, but it never shrinks from the initial value. This improvement significantly minimizes the number of collisions at early stages, as well as the number of possibilities we have to consider during reconfiguration.

The above logic of vtblmap always chooses the configuration that renders no conflicts, when such a configuration is possible during recomputation of optimal parameters. When this is not possible, it is natural to prefer collisions to happen on less-frequent vtbl-pointers.

We studied the frequency of vtbl-pointers that come through various match statements of a C++ pretty-printer that we implemented on top of the Pivot framework [48] using our pattern-matching library. We ran the pretty-printer on a set of C++ standard library headers and then ranked all the classes from the most-frequent to the least-frequent ones, on average. The resulting probability distribution is shown with a thicker line in Figure8.

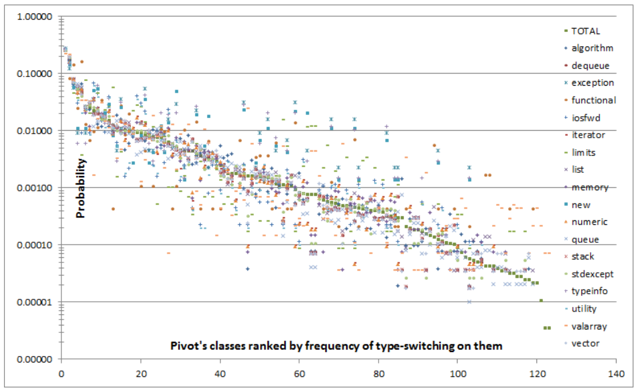

Figure 8. Probability distribution of various nodes in Pivot framework

Note that Y-Axis is using logarithmic scale, suggesting that the resulting probability has power-law distribution. This is likely to be a specifics of our application, nevertheless, the above picture demonstrates that frequency of certain classes can be larger than the overall frequency of all the other classes. In our case, the two most frequent classes were representing the use of a variable in a program, and their combined frequency was larger than the frequency of all the other nodes. Naturally, we would like to avoid conflicts on such classes in the cache, when possible.

Let us assume that a given vtblmap contains a set of vtbl pointers $V = \{v_1, ..., v_n\}$ with known probabilities p_i of occuring. For a cache of size 2^k and a shift by l bits we get a cache-indexing function $f_{lk} : V \to [0..2^k - 1]$ defined as $f_{lk}(v_i) = (v_i \gg l)\&(2^k - 1)$. To calculate the probability of conflict for a given l and k parameters, let us consider j^{th} cache cell and a subset $V_{lk}^j = \{v \in V | f_{lk}(v) = j\}$. When the size of this subset $m = |V_{lk}^j|$ is greater than 1, we have a potential conflict as subsequent request for a vtbl pointer v'' might be different from the vtbl pointer v' currenly stored in the cell j. Within the cell only the probability of not having a conflict is the probability of both values v'' and v' be the same:

$$P(v''=v') = \sum_{v_i \in V_{lk}^j} P(v''=v_i)P(v'=v_i) = \sum_{v_i \in V_{lk}^j} P^2(v_i|V_{lk}^j) =$$

$$= \sum_{v_i \in V_{lk}^j} \frac{P^2(v_i)}{P^2(V_{lk}^j)} = \sum_{v_i \in V_{lk}^j} \frac{p_i^2}{(\sum\limits_{v_{i'} \in V_{lk}^j} p_{i'})^2} = \frac{\sum\limits_{v_i \in V_{lk}^j} p_i^2}{(\sum\limits_{v_i \in V_{lk}^j} p_i)^2}$$

The probability of having a conflict among the vtbl pointers of a given cell is thus one minus the above value:

$$P(v'' \ne v') = 1 - \frac{\sum\limits_{v_i \in V_{lk}^j} p_i^2}{(\sum\limits_{v_i \in V_{lk}^j} p_i)^2}$$

To obtain probability of conflict given any vtbl pointer and not just the one from a given cell we need to sum up the above probabilities of conflict within a cell multiplied by the probability of vtbl pointer fall into that cell:

$$P_{lk}^{conflict} = \sum_{j=0}^{2^k-1} P(V_{lk}^j)(1 - \frac{\sum\limits_{v_i \in V_{lk}^j} p_i^2}{(\sum\limits_{v_i \in V_{lk}^j} p_i)^2}) =$$

$$= \sum_{j=0}^{2^k-1} (\sum_{v_i \in V_{lk}^j} p_i)(1 - \frac{\sum\limits_{v_i \in V_{lk}^j} p_i^2}{(\sum\limits_{v_i \in V_{lk}^j} p_i)^2})$$

Our reconfiguration algorithm then iterates over possible values of l and k and chooses those that minimize the overal probability of conflict $P_{lk}^{conflict}$. The only data still missing are the actual probabilities p_i used by the above formula. They can be approximated in many different ways.

Besides probability distribution on all the tests, Figure 8 shows probabilities of a given node on each of the tests. The X-Axis in this case represents the ordering of all the nodes according to their overall rank of all the tests combined. As can be seen from the picture, the shape of each specific test's distribution still mimics the overal probability distribution. With this in mind we can simply let the user assign probabilities to each of the classes in the hierarchy and use these values during reconfiguration. The practical problem we came accross with this solution was that we wanted these probabilities be inheritable as Pivot separates interface and implementation classes and we prefered the user to define them on interfaces rather than on implementation classes. The easiest way to do so wast to write a dedicated function that would return the probabilities using a match statement. Unfortunately such a function will introduce a lot of overhead as it will ideally only be used very few times (since we try to minimize the amount of reconfiguration) and thus not be using memoized jumps but rather slow cascading-if.

A simpler and likely more precise way of estimating p_i would be to count frequencies of each vtbl pointers directly inside the vtblmap. This introduces an overhead of an increment into the critical path of execution, but according to our tests was only degrading the overal performance by 1-2%. Instead, it was compensating with a smaller amount of conflicts and thus a potential gain of performance. We leave the choice of whether the library should count frequencies of each vtbl pointer to the user of the library as the concrete choice may be to advantage on some class hierarchies and to disadvantage on others.

Figure 9 compares the amount of collisions when frequency information is and is not used. The data was gathered from 312 tests on multiple match statements present in Pivot's C++ pretty printer when it was ran over standard library headers. In 122 of these test both schemes had 0 conflicts and these tests are thus not shown on the graph. The remaining tests where ranked by the amount of conflicts in the scheme that does not utilize frequency information.

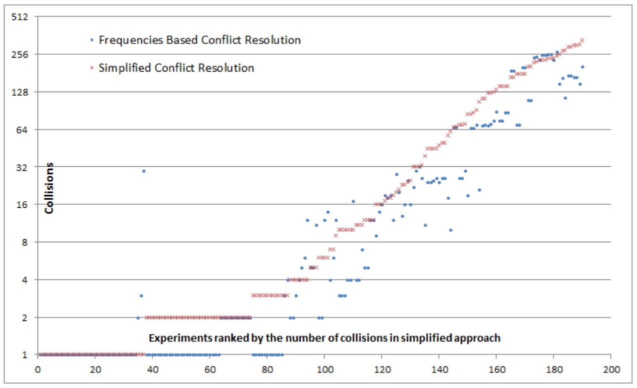

Figure 9. Decrease in number of collisions when probabilities of nodes are taken into account

As can be seen from the graph, both schemes render quite low amount of collisions given that there was about 57000 calls in the rightmost test having the largest amount of conflicts. Taking into account that the Y-axis has logarithmic scale, the use of frequency information in many cases decreased the amount of conflicts by a factor of 2. The handfull of cases where the use of frequency increased the number of conflicts can be explained by the fact that the optimal values are not recomputed after each conflict, but after several conflicts and only if the amount of vtbl pointers in the vtblmap increased. These extra conditions sacrify optimality of parameters at any given time for the amount of times they are recomputed. By varying the number of conflicts we are willing to tolerate before reconfiguration we can decrease the number of conflicts by increasing the amount of recomputations and vise versa. From our experience, however, we saw that the drop in the number of conflicts was not translating into a proportional drop in execution time, while the amount of reconfigurations was proportional to the increase in execution time. This is why we choose to tolerate a relatively large amount of conflicts before recomputation just to keep the amount of recomputations low.

10. Solution for Tagged Classes

The memoization device outlined in §9.2 can, in principle, also be applied to tagged classes. The dynamic cast will be replaced by a small compile-time template meta-program that checks whether the class associated with the given tag is derived from the target type of the case clause. If so, a static cast can be used to obtain the offset.

Despite its straightforwardness, we felt that it should be possible to do better than the general solution, given that each class is already identified with a dedicated constant known at compile time.

As we mentioned in §8.2, the nominal subtyping of C++ effectively gives every class multiple types. The idea is thus to associate with the type not only its most derived tag, but also the tags of all its base classes. In a compiler implementation such a list can be stored inside the virtual table of a class, while in our library solution it is shared between all the instances with the same most derived tag in a less efficient global map, associating the tag to its tag list.

The list of tags is topologically sorted according to the subtyping relation and terminates with a dedicated value distinct from all the tags. We call such a list a *Tag Precedence List* (TPL) as it resembles the *Class Precedence List* (CPL) of object-oriented descendants of Lisp (e.g. Dylan, Flavors, LOOPS, and CLOS) used there for *linearization* of class hierarchies. The classes in CPL are ordered from most specific to least specific with siblings listed in the *local precedence order* – the order of the direct base classes used in the class definition. TPL is just an implementation detail and the only reason we distinguish TPL from CPL is that in C++ classes are often separated into interface and implementation classes and it might so happen that the same tag is associated by the user with an interface and several implementation classes.

A type switch below, built on top of a hierarchy of tagged classes, proceeds as a regular switch on the subject's tag. If the jump succeeds, we found an exact match; otherwise, we get into a default clause that obtains the next tag in the tag precedence list and jumps back to the beginning of the switch statement for a rematch:

```
    const size_t* taglist = 0;
          size_t attempt = 0;
          size_t tag = object→ tag;
ReMatch:
    switch (tag)
    {
    default:
        if (!taglist)
            taglist = get_taglist(object→ tag);
        tag = taglist[++attempt];
        goto ReMatch;
    case end_of_list: break;
    case bindings⟨$D_1$⟩::kind_value: $s_1$;break;
    ...
    case bindings⟨$D_n$⟩::kind_value: $s_n$;break;
    }
```

The above structure, which we call *TPL Dispatcher*, lets us dispatch to case clauses of the most derived class with an overhead of initializing two local variables, compared to a switch on a sealed hierarchy. Dispatching to a case clause of a base class will take time roughly proportional to the distance between the matched base class and the most derived class in the inheritance graph. When none of the base class tags was matched, we will necessarily reach the end_of_list marker in the tag precedence list and thus exit the loop.

Our library automatically builds the function get_taglist based on the BC or BCS specifiers that the user specifies in bindings (§4.1).

To make the behavior clos

```
    if (is_derived_from⟨$D_i$⟩(object))
    {
case bindings⟨$D_i$⟩::kind_value:
        auto matched = static_cast⟨$D_i$⟩(object);
    s_i;
}
```

11. (Ab)using Exceptions for Type Switching

Several authors had noted the relationship between exception handling and type switching before [2, 25]. Not surprisingly, the exception handling mechanism of C++ can be abused to implement the first-fit semantics of a type switch statement. The idea is to harness the fact that catch-handlers in C++ essentially use first-fit semantics to decide which one is going to handle a given exception. The only problem is to raise an exception with a static type equal to the dynamic type of the subject.

To do this, we employ the *polymorphic exception* idiom [55] that introduces a virtual function **virtual void** raise() **const** = 0; into the base class, overridden by each derived class in syntactically the same way: **throw** ∗**this**;. The *Match*-statement then simply calls raise on its subject, while case clauses are turned into catch-handlers. The exact name of the function is not important, and is communicated to the library as *raise selector* with RS specifier in the same way *kind selector* and *class members* are (§4.1). The raise member function can be seen as an analog of the accept member function in the visitor design pattern, whose main purpose is to discover subject's most specific type. The analog of a call to visit to communicate that type is replaced, in this scheme, with exception unwinding mechanism.

Just because we can, it does not mean we should abuse the exception handling mechanism to give us the desired control flow. In the table-driven approach commonly used in high-performance implementations of exception handling, the speed of handling an exception is sacrificed to provide a zero execution-time overhead for when exceptions are not thrown [51]. Using exception handling to implement type switching will reverse the common and exceptional cases, significantly degrading performance. As can be seen in Figure4, matching the type of the first case clause with polymorphic exception approach takes more than 7000 cycles and then grows linearly (with the position of case clause in the match statement), making it the slowest approach. The numbers illustrate why exception handling should only be used to deal with exceptional and not common cases.

Despite its total inpracticality, the approach fits well into our unified syntax (§12) and gave us a very practical idea of harnessing a C++ compiler to do *redundancy checking* at compile time.

11.1 Redundancy Checking

As discussed in §2, redundancy checking is only applicable to first-fit semantics of the match statement, and warns the user of any case clause that will never be entered because of a preceding one being more general.

We provide a library-configuration flag, which, when defined, effectively turns the entire match statement into a try-catch block with handlers accepting the target types of the case clauses. This forces the compiler to give warning when a more general catch handler preceds a more specific one effectively performing redundancy checking for us, e.g.:

```
filename.cpp(55): warning C4286: 'ipr::Decl*' : is caught by
                  base class ('ipr::Stmt*') on line 42
```

Note that the message contains both the line number of the redundant case clause (55) and the line number of the case clause that makes it redundant (42).

Unfortunately, the flag cannot be always enabled, as the case labels of the underlying switch statement have to be eliminated in order to render a syntactically correct program. Nevertheless, we found the redundancy checking facility of the library extremely useful when rewriting visitor-based code: even though the order of overrides in a visitor's implementation does not matter, for some reason more general ones were inclined to happen before specific ones in the code we looked at. Perhaps programmers are inclined to follow the class declaration order when defining and implementing visitors.

A related *completeness checking* – test of whether a given match statement covers all possible cases – needs to be reconsidered for extensible data types like classes, since one can always add a new variant to it. Completeness checking in this case may simply become equivalent to ensuring that there is either a default clause in the type switch or a clause with the static type of a subject as a target type. In fact, our library has an analog of a default clause called *Otherwise*-clause, which is implemented under the hood exactly as a regular case clause with the subject's static type as a target type.

12. Unified Syntax

The discussion in this subsection will be irrelevant for a compiler implementation, nevertheless we include it because some of the challenges we came accross as well as techniques we used to overcome them might show up in other active libraries. The problem is that working in a library setting, the toolbox of properties we can automatically infer about user's class hierarchy, match statement, clauses in it, etc. is much more limited than the set of properties a compiler can infer. On one side such additional information may let us generate a better code, but on the other side we understand that it is important not to overburden the user's syntax with every bit of information she can possibly provide us with to generate a better code. Some examples of information we can use to generate a better code even in the library setting include:

- Encoding we are dealing with (§3)
- Shape of the class hierarchy: flat/deep, single/multiple inheritance etc.
- The amount of clauses in the match statement
- Presense of Otherwise clause in the match statement
- Presence of extensions in dynamically linked libraries

We try to infer the information when we can, but otherwise resort to a usually slower default that will work in all or most of the cases. The major source of inefficiency comes from the fact that macro resolution happens before any meta-programming techniques can be employed and thus the macros have to generate a syntactic structure that can essentially handle all the cases as opposed to the exact case. Each of the macros involved in rendering the syntactic structure of a match statement (e.g. *Match*, *Case*, *Otherwise*) have a version identified with a suffix that is specific to a combination of encoding and shape of the class hierarchy. By default the macros are resolved to a unified version that infers encoding with a template meta-program, but this resolution can be overriden with a configuration flag for a more specific version when all the match statements in user's program satisfy the requirements of that version. The user can also pin-point specific match statement with the most applicable version, but we discourage such use as performance differences are not big enough to justify the exposure of details.

To better understand what is going on, consider the following examples. Case labels for polymorphic base class encoding can be arbitrary, but preferably sequential numbers, while the case labels for tagged class and discriminated union encodings are the actual kind values associated with concrete variants. Discriminated union and tagged class encodings can use both types (views in case of unions) and kind values to identify the target variant, while polymorphic base class encoding can only use types for that. The latter encoding requires allocation of a static vtblmap in each match statement, not needed by any other encoding, while tagged class encoding on non-flat hierarchy requires the use of default label of the generated switch statement as well as a dedicated case label distinct from all kind values (§10). When merging these and other requirements into a syntactic structure of a unified version capable of handling any encoding we essentially always have to reserve the use of default label (and thus not use it to generate *Otherwise*-clause), allocate an extra dedicated case label, introduce a loop over base classes used by tagged class encoding etc. This is a clear overhead for handling of a discriminated union encoding whose syntactic structure only involves a simple switch over kind values and default label to implement *Otherwise*. To minimize the effects of this overhead we rely on compiler's optimizer to inline calls specific to each encoding and either remove branching on conditions that will always be true after inlining or elminate dead code on conditions that will always be false after inlining. Luckily for us today's compilers do a great job in doing just that, rendering our unified version only slightly less efficient than the specialized ones. These differences can be best seen in Figure10 under corresponding entries of *Unified* and *Specialized* columns.

13. Memoized Dynamic Cast

We saw in Corollary 1 that the results of **dynamic_cast** can be reapplied to a different instance from within the same subobject. This leads to simple idea of memoizing the results of **dynamic_cast** and then using them on subsequent casts. In what follows we will only be dealing with the pointer version of the operator since the version on references that has a slight semantic difference can be easily implemented in terms of the pointer one.

The **dynamic_cast** operator in C++ involves two arguments: a value argument representing an object of a known static type as well as a type argument denoting the runtime type we are querying. Its behavior is twofold: on one hand it should be able to determine when the object's most derived type is not a subtype of the queried type (or when the cast is ambiguous), while on the other it should be able to produce an offset by which to adjust the value argument when it is.

We mimic the syntax of **dynamic_cast** by defining:

```
template ⟨typename T, typename S⟩
inline T memoized_cast(S* p);
```

which lets the user replace all the uses of **dynamic_cast** in the program with memoized_cast with a simple:

```
#define dynamic_cast memoized_cast
```

It is important to stress that the offset is not a function of the source and target types of the **dynamic_cast** operator, which is why we cannot simply memoize the outcome inside the individual instantiations of memoized_cast. The use of repeated multiple inheritance will result in classes having several different offsets associated with the same pair of source and target types depending on which subobject the cast is performed from. According to corollary 1, however, it is a function of target type and the value of the vtbl-pointer stored in the object, because the vtbl-pointer uniquely determines the subobject within the most derived type. Our memoization of the results of **dynamic_cast** should thus be specific to a vtbl-pointer and the target type.

The easiest way to achieve this would be to use a dedicated global vtblmap⟨std::ptrdiff_t⟩ (§9.3.1) per each instantiation of the memoized_cast. This, however, will create an unnecessarily large amount of vtblmap structures, many of which will be duplicating information and repeating the work already done. This will happen because instantiations of memoized_cast with same target but different source types can share their vtblmap structures since vtbl pointers of different source types are necessarily different according to Theorem 1.

Even though the above solution can be easily improved to allocates a single vtblmap per target type, an average application might have a lot of different target types. This is especially true for applications that will use our Match statement since we use **dynamic_cast** under the hood in each case clause. Indeed our C++ pretty printer was creating 160 vtblmaps of relatively small size each, which was increasing the executable size quite significantly because of numerous instantiations as well as noticably slowed down the compilation time.

To overcome the problem we turn each target type into a runtime instantiation index of the type and allocate a single vtblmap⟨std::vector⟨std::ptrdiff_t⟩⟩ that associates vtbl pointers with a vector of offsets indexed by target type. The slight performance overhead that is brought by this improvement is specific to our library solution and would not be present in a compiler implementaion. Instead we get a much smaller memory footrpint, which can be made even smaller once we recognize the fact that global type indexing may effectively enumerate target classes that will never appear in the same Match statement. This will result in entries in the vector of offsets that are never used.

Our actual solution uses separate indexing of target types for each source type they are used with, and also allocates a different vtblmap⟨std::vector⟨std::ptrdiff_t⟩⟩ for each source type. This lets us minimize unused entries within offset vectors by making sure only the plausible target types for a given source type are indexed. This solution should be suitable for most applications since we expect to have a fairly small number of source types for the **dynamic_cast** operator and a much larger number of target types. For the unlikely case of a small number of target types and large number of source types we allow the user to revert to the default behavior with a library configuration switch that allocates a single vtblmap per target type as we have already discussed above.

The use of memoized_cast to implement the *Match*-statement potentially reuses the results of **dynamic_cast** computations across multiple independent match statements. This allows leveraging the cost of the expensive first call with a given vtbl-pointer even further across all the match statements inside the program. The above define, with which a user can easily turn all dynamic casts into memoized casts, can be used to speed-up existing code that uses dynamic casting without any refactoring overhead.

14. Evaluation

In this section, we evaluate the performance of our solution in comparison to its de-facto contender – the visitor design pattern. We also compare performance of some typical use cases expressed with our solution and OCaml.

Our evaluation methodology consists of several benchmarks that we believe represent various possible uses of objects inspected with either visitors or pattern matching.

The *repetitive* benchmark performs multiple calls on different objects of the same most derived type. This scenario happens in object-oriented setting when a group of polymorphic objects is created and passed around (e.g. numerous particles of a given kind in a particle simulation system). We include it because double dispatch becomes about twice faster (27 vs. 53 cycles) in this scenario compared to others due to cache and call target prediction mechanisms.

The *sequential* benchmark effectively uses an object of each derived type only once and then moves on to an object of a different type. The cache is typically reused the least in this scenario. The scenario is typical of lookup tables, where each entry is implemented with a different derived class.

		G++/32 on Windows Laptop				MS Visual C++/32				MS Visual C++/64			
	Syntax	Unified		Specialized		Unified		Specialized		Unified		Specialized	
	Encoding	**Open**	**Tag Union**	**Open**	**Tag Union**	**Open**	**Tag Union**	**Open**	**Tag Union**	**Open**	**Tag Union**	**Open**	**Tag Union**
	Repetitive	55%	116%	55%	216%	1%	35%	1%	133%	**33%**	**8%**	**27%**	38%
	Sequential	1%	43%	3%	520%	**10%**	**5%**	**8%**	59%	**43%**	**0%**	**45%**	3%
	Random	**0%**	29%	1%	542%	**1%**	**6%**	**1%**	25%	**47%**	**5%**	**44%**	12%
Forward	Repetitive	67%	88%	67%	79%	10%	**3%**	10%	**7%**	**24%**	**61%**	**9%**	**79%**
Forward	Sequential	87%	250%	90%	259%	**0%**	**11%**	**0%**	**10%**	**36%**	**25%**	133%	**35%**
Forward	Random	28%	32%	27%	31%	14%	**8%**	14%	**7%**	**24%**	**24%**	**23%**	**25%**

		G++/32 on Linux Desktop				MS Visual C++/32 with PGO				MS Visual C++/64 with PGO			
	Syntax	Unified		Specialized		Unified		Specialized		Unified		Specialized	
	Encoding	**Open**	**Tag Union**	**Open**	**Tag Union**	**Open**	**Tag Union**	**Open**	**Tag Union**	**Open**	**Tag Union**	**Open**	**Tag Union**
	Repetitive	19%	54%	16%	124%	4%	61%	4%	124%	**14%**	20%	**0%**	47%
	Sequential	62%	109%	56%	640%	**9%**	13%	3%	34%	2%	34%	1%	14%
	Random	57%	97%	56%	603%	**17%**	18%	**18%**	43%	**27%**	7%	**27%**	16%
Forward	Repetitive	36%	45%	33%	53%	10%	16%	10%	31%	**5%**	**5%**	**6%**	**9%**
Forward	Sequential	53%	77%	55%	86%	153%	168%	153%	185%	130%	132%	145%	118%
Forward	Random	76%	82%	78%	88%	19%	11%	18%	24%	**5%**	**2%**	**6%**	**10%**
						Windows Laptop							

Figure 10. Relative performance of type switching versus visitors. Numbers in regular font (e.g. 67%), indicate that our type switching is faster than visitors by corresponding percentage. Numbers in bold font (e.g. **14%**), indicate that visitors are faster by corresponding percentage.

The *random* benchmark is the most representative as it randomly makes calls on random objects, which will probably be the most common usage scenario in the real world.

The *forwarding* benchmark is not a benchmark on its own, but rather a combinator that can be applied to any of the above scenarios. It refers to the common technique used by visitors where, for class hierarchies with multiple levels of inheritance, the visit method of a derived class will provide a default implementation of forwarding to its immediate base class, which, in turn, may forward it to its base class, etc. The use of forwarding in visitors is a way to achieve substitutability, which in type switches corresponds to the use of base classes in the case clauses. This approach is used in Pivot, whose AST hierarchy consists of 154 node kinds, of which only 5 must be handled, while the rest will forward to them when visit for them was not overriden.

The class hierarchy for non-forwarding test was a flat hierarchy with 100 derived classes, encoding an algebraic data type. The class hierarchy for forwarding tests had two levels of inheritance with 5 intermediate base classes and 95 derived ones.

Each benchmark was tested with either *unified* or *specialized* syntax, each of which included tests on polymorphic (*Open*) and tagged (*Tag*) encodings. Specialized syntax avoids generating unnecessary syntactic structure used to unify syntax, and thus produces faster code. We include it in our results because a compiler implementation of type switching will only generate the best suitable code.

The benchmarks were executed in the following configurations refered to as *Linux Desktop* and *Windows Laptop* respectively:

- Dell Dimension® desktop with Intel® Pentium® D (Dual Core) CPU at 2.80 GHz; 1GB of RAM; Fedora Core 13
  - G++ 4.4.5 executed with -O2
- Sony VAIO® laptop with Intel® Core™i5 460M CPU at 2.53 GHz; 6GB of RAM; Windows 7 Professional
  - G++ 4.5.2 / MinGW executed with -O2; x86 binaries
  - MS Visual C++ 2010 Professional x86/x64 binaries with profile-guided optimizations

The code on the critical path of our type switch implementation benefits significantly from branch hinting as some branches are much more likely than others. We use the branch hinting facilities of GCC to tell the compiler which branches are more likely, but, unfortunately, Visual C++ does not have similar facilities. The official way suggested by Microsoft to achieve the same effect is to use *Profile-Guided Optimization* and let the compiler gather statistics on each branch. This is why the result for Visual C++ reported here are those obtained with profile-guided optimizations enabled. The slightly less-favorable-for-us results without profile-guided optimizations can be found in the accompanying technical report [3].

We compare the performance of our solution relative to the performance of visitors in Figure 10. The values are given as percentages of performance increase against the slower technique. Numbers in regular font represent cases where our type switching was faster than visitors were. Numbers in bold indicate cases where visitors were faster.

From the numbers, we can see that type switching wins by a good margin in the presence of at least one level of forwarding on visitors. Using type switching on closed hierarchies is also a definite winner.

From the table it may seem that Visual C++ is generating not as good code as GCC does, but remember that these numbers are relative, and thus the ratio depends on both the performance of virtual calls and the performance of switch statements. Visual C++ was generating faster virtual function calls, while GCC was generating faster switch statements, which is why their relative performance seem to be much more favorable for us in the case of GCC.

Similarly the code for x64 is only slower relatively: the actual time spent for both visitors and type switching was smaller than that for x86, but it was much smaller for visitors than type switching, which resulted in worse relative performance.

14.1 Vtable Pointer Memoization vs. TPL Dispatcher

With a few exceptions for x64, it can be seen from Figure 10 that the performance of the TPL dispatcher (the Tag column) dominates the performance of the vtable pointer memoization approach (the Open column). We believe that the difference, often significant, is the price one pays for the true openness of the vtable pointer memoization solution.

Unfortunately, the TPL dispatcher is not truly open. The use of tags, even if they would be allocated by compiler, may require integration efforts to ensure that different DLLs have not reused the same tags. Randomization of tags, similar to a proposal of Garrigue [21], will not eliminate the problem and will surely replace jump tables in switches with decision trees. This will likely significantly degrade the numbers for the Tag column of Figure 10, since the tags in our experiments were all sequential.

Besides, the TPL dispatcher approach relies on static cast to obtain the proper reference once the most specific case clause has been found. As we described in §9.3, this has severe limitations in the presence of multiple inheritance, and thus is not as versatile as the other solution. Overcoming this problem will either require the use of **dynamic_cast** or techniques similar to those we used in vtable pointer memoization. This will likely degrade performance numbers for the Tag column even further.

Note also that the vtable pointer memoization approach can be used to implement both first-fit and best-fit semantics, while the TPL dispatcher is only suitable for best-fit semantics. Their complexity guarantees also differ: vtable pointer memoization is constant on average, and slow on the first call. Tag list approach is logarithmic in the size of the class hierarchy on average (assuming a balanced hierarchy), including on the first call.

14.2 Comparison with OCaml

We now compare our solution to the built-in pattern-matching facility of OCaml [31]. In this test, we timed a small OCaml application performing our sequential benchmark on an algebraic data type of 100 variants. Corresponding C++ applications were working with a flat class hierarchy of 100 derived classes. The difference between the C++ applications lies in the encoding (Open/Tag/Kind) and the syntax (Unified/Special) used. Kind encoding is the same as Tag encoding, but it does not require substitutability, and thus can be implemented with a direct switch on tags. It is only supported through specialized syntax in our library as it differs from the Tag encoding only semantically.

The optimized OCaml compiler `ocamlopt.opt` that we used to compile the code can be based on different toolsets on some platforms, e.g. Visual C++ or GCC on Windows. To make the comparison fair we had to make sure that the same toolset was used to compile the C++ code. We ran the tests on both of the machines described above using the following configurations:

- The tests on a Windows 7 laptop were all based on the *Visual C++ toolset* and used `ocamlopt.opt` version 3.11.0.
- The tests on a Linux desktop were all based on the *GCC toolset* and used `ocamlopt.opt` version 3.11.2

The timing results presented in Figure 11 are averaged over 101 measurements and show the number of seconds it took to perform a million decompositions within our sequential benchmark.

We can see that the use of specialized syntax on a closed/sealed hierarchy can match the speed of, and even be four times faster than, the code generated by the native OCaml compiler. Once we go for an open solution, we become about 30-50% slower.

14.3 Qualitative Comparison

For this experiment we have reimplemented a visitor based C++ pretty printer for Pivot's IPR using our pattern matching library. Most of the rewrite was performed by sed-like replaces that converted visit methods into respective case-clauses. In several cases we had to manually reorder case-clauses to avoid redundancy as visit-methods for base classes were typically coming before the same for derived, while for pattern matching we needed them to

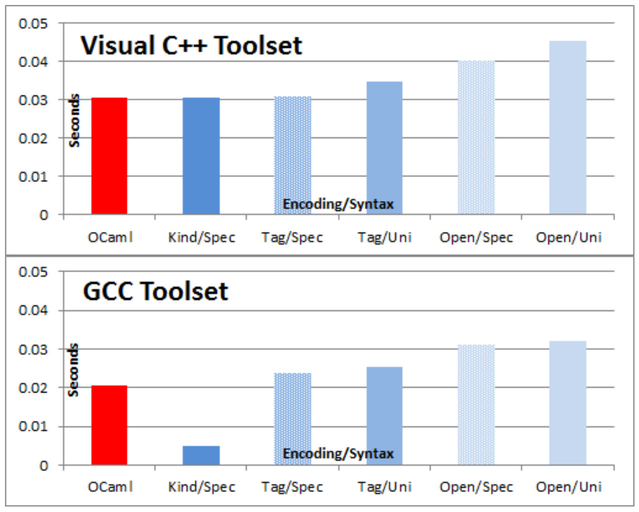

Figure 11. Performance comparison of various encodings and syntax against OCaml code

come after. Redundancy checking support in the library discussed in §11.1 was invaluable in finding out all such cases.

During this refactoring we have made several simplifications that became obvious in pattern-matching code, but were not in visitors code because of control inversion. Simplifications that were applicable to visitors code were eventually integrated into visitors code as well to make sure we do not compare algorithmically different code. In any case we were making sure that both approaches regardless of simplifications were producing byte-to-byte the same output as the original pretty printer we started from.

The size of executable for pattern matching approach was smaller than that for visitors. So was also the source code. We extracted from both sources the functionality that was common to them and placed it in a separate translation unit to make sure it does not participate in the comparison. We kept all the comments however that were eqaully applicable to code in either approach.

Note that the visitors involved in the pretty printer above did not use forwarding: since all the C++ constructs were handled by the printer, every visit-method was overriden from those statically possible based on the static type of the argument.

In general from our rewriting experience we will not recommend rewriting existing visitor code with pattern matching for the simple reason that pattern matching code will likely follow the structure already set by the visitors. Pattern matching was most effective when writing new code, where we could design the structure of the code having the pattern matching facility in our toolbox.

15. Related Work

There are two main approaches to compiling pattern matching code: the first is based on *backtracking automata* and was introduced by Augustsson[], the second is based on *decision trees* and is attributed in the literature to Dave MacQueen and Gilles Kahn in their implementation of Hope compiler []. Backtracking approach usually generates smaller code, while decision tree approach produces faster code by ensuring that each primitive test is only performed once. Neither of the approaches addresses specifically type patterns or type switching and simply assumes presence of a primitive operation capable of performing type tests.

Memoization device we proposed is not specifically concerned with compiling pattern matching and can be used independently. In particular it can be combined with either backtracking or decision tree approaches to avoid subsequent decisions on datum that has already been seen.

Extensible Visitors with Default Cases [64, §4.2] attempt to solve the extensibility problem of visitors; however, the solution, after remapping it onto C++, has problems of its own. The visitation interface hierarchy can easily be grown linearly (adding new cases for the new classes in the original hierarchy each time), but independent extensions by different authorities require developer's intervention to unify them all, before they can be used together. This may not be feasible in environments that use dynamic linking. To avoid writing even more boilerplate code in new visitors, the solution would require usage of virtual inheritance, which typically has an overhead of extra memory dereferencing. On top of the double dispatch already present in the visitor pattern, the solution will incur two additional virtual calls and a dynamic cast for each level of visitor extension. Additional double dispatch is incurred by forwarding of default handling from a base visitor to a derived one, while the dynamic cast is required for safety and can be replaced with a static cast when the visitation interface is guaranteed to be grown linearly (extended by one authority only). Yet another virtual call is required to be able to forward computations to subcomponents on tree-like structures to the most derived visitor. This last function lets one avoid the necessity of using the heap to allocate a temporary visitor through the *Factory Design Pattern* [20] used in the *Extensible Visitor* solution originally proposed by Krishnamurti, Felleisen and Friedman [30].

In order to address the expression problem in Haskell, Löh and Hinze proposed to extend its type system with open data types and open functions [36]. Their solution allows the user to mark top-level data types and functions as open and then provide concrete variants and overloads anywhere in the program. Open data types are extensible but not hierarchical, which largely avoids the problems discussed here. The semantics of open extension is given by transformation into a single module, where all the definitions are seen in one place. This is a significant limitation of their approach that prevents it from being truly open, since it essentially assumes a whole-program view, which excludes any extension via DLLs. As is the case with many other implementations of open extensions, the authors rely on the closed world for efficient implementation: in their implementation, *"data types can only be entirely abstract (not allowing pattern matching) or concrete with all constructors with the reason being that pattern matching can be compiled more efficiently if the layout of the data type is known completely"*. The authors also believe that *there are no theoretical difficulties in lifting this restriction, but it might imply a small performance loss if closed functions pattern match on open data types*. Our work addresses exactly this problem, showing that it is not only theoretically possible but also practically efficient and in application to a broader problem.

Polymorphic variants in OCaml [21] allow the addition of new variants later. They are simpler, however, than object-oriented extensions, as they do not form subtyping between variants themselves, but only between combinations of them. This makes an important distinction between *extensible sum types* like polymorphic variants and *extensible hierarchical sum types* like classes. An important property of extensible sum types is that each value of the underlying algebraic data type belongs to exactly one disjoint subset, tagged with a constructor. The *nominative subtyping* of object-oriented languages does not usually have this disjointness making classes effectively have multiple types. In particular, the case of disjoint constructors can be seen as a degenerated case of a flat class hierarchy among the multitude of possible class hierarchies.

Tom is a pattern-matching compiler that can be used together with Java, C or Eiffel to bring a common pattern matching and term rewriting syntax into the languages[40]. It works as a preprocessor that transforms syntactic extensions into imperative code in the target language. Tom is quite transparent as to the concrete target language used and can potentially be extended to other target languages besides the three supported now. In particular, it never uses any semantic information of the target language during the compilation process and it does not inspect nor modify the source language part (their preprocessor is only aware of parenthesis and block delimiters of the source language). Tom has a sublanguage called Gom that can be used to define algebraic data types in a uniform mannaer, which their preprocessor then transforms into concrete definitions in the target language. Alternatively, the user can provide mappings to his own data structures that the preprocessor will use to generate the code.

In comparison to our approach Tom has much bigger goals. The combination of pattern matching, term rewriting and strategies turns Tom into a tree-transformation language similar to Stratego/XT, XDuce and others. The main accent is made on expressivity and the speed of development, which makes one often wonder about the run-time complexity of the generated code. Tom's approach is also prone to general problems of any preprocessor based solution[54, §4.3]. For example, when several preprocessors have to be used together, each independent extension may not be able to understand the other's syntax, making it impossible to form a toolchain. A library approach we follow avoids most of these problems by relying only on a standard C++ compiler. It also lets us employ semantics of the language within patterns: e.g. our patterns work directly on underlying user-defined data structures, largely avoiding abstraction penalties. A tighter integration with the language semantics also makes our patterns first-class citizens that can be composed and passed to other functions. The approach we take to type switching can also be used by Tom's preprocessor to implement type patterns efficiently – similarly to other object-oriented languages, Tom's handling of them is based on highly inefficient `instanceof` operator and its equivalents in other languages.

Pattern matching in Scala [42] also allows type patterns and thus type switching. The language supports extensible and hierarchical data types, but their handling in a type switching constructs varies. Sealed classes are handled with an efficient switch over all tags, since sealed classes cannot be extended. Classes that are not sealed are similarly approached with a combination of an `InstanceOf` operator and a decision tree [17].

There has been previous attempts to use pattern matching with the Pivot framework that we used to experiment with our library. In his dissertation, Pirkelbauer devised a pattern language capable of representing various entities in a C++ program. The patterns were then translated with a tool into a set of visitors implementing the underlying pattern matching semantics[46]. Earlier, Cook et al used expression templates to implement a query language for Pivot's Internal Program Representation [8]. While their work was built around a concrete class hierarchy letting them put some semantic knowledge about concrete classes into the The principal difference of their work from this work is that authors were essentially creating a pattern matcher for a given class hierarchy and thus could take the semantics of the entities represented by classes in the hierarchy into account. Our approach is parametrized over class hierarchy and thus provides a rather lower level pattern-matching functionality that lets one simplify work with that hierarchy. One can think of it as a generalized dynamic_cast. To be continued...

16. Future Work

In the future we would like to provide an efficient multi-threaded implementation of our library as currently it relies heavily on static variables and global state, which will have problems in a multi-threaded environment.

The match statement that we presented here deals with only one subject at the moment, but we believe that our memoization device, along with the vtable pointer memoization technique we presented, can cope reasonably efficiently with multiple subjects. Their support will make our library more general by addressing asymmetric multiple dispatch.

We would also like to experiment with other kinds of cache indexing functions in order to decrease the frequency of conflicts, especially those coming from the use of dynamically-linked libraries.

Containers as described by the standard C++ library do not have the implicit recursive structure present in lists, sequences and other recursive data structures of functional languages. Viewing them as such with view will likely have a significant performance overhead, not usually affordable in the kind of applications C++ is used for. We therefore would like to experiment with some pattern matching alternatives that will let us work with STL containers efficiently yet expressively as in functional languages.

17. Conclusions

Type switching is an open alternative to visitor design pattern that overcomes the restrictions, inconveniences, and difficulties in teaching and using, typically associated with it. Our implementation of it comes close or outperforms the visitor design pattern, which is true even in a library setting using a production-quality compiler, where the performance base-line is already very high.

We describe three techniques that can be used to implement type switching, type testing, pattern matching, predicate dispatching, and other facilities that depend on the run-time type of an argument as well as demonstrate their efficiency.

The *Memoization Device* is an optimization technique that maps run-time values to execution paths, allowing to take shortcuts on subsequent runs with the same value. The technique does not require code duplication and in typical cases adds only a single indirect assignment to each of the execution paths. It can be combined with other compiler optimizations and is particularly suitable for use in a library setting.

The *Vtable Pointer Memoization* is a technique based on memoization device that employs uniqueness of virtual table pointers to not only speed up execution, but also properly uncover the dynamic type of an object. This technique is a backbone of our fast type switch as well as memoized dynamic cast optimization.

The *TPL Dispatcher* is yet another technique that can be used to implement best-fit type switching on tagged classes. The technique has its pros and cons in comparison to vtable pointer memoization, which we discuss in the paper.

These techniques can be used in a compiler and library setting, and support well separate compilation and dynamic linking. They are open to class extensions and interact well with other C++ facilities such as multiple inheritance and templates. The techniques are not specific to C++ and can be adopted in other languages for similar purposes.

Using the above techniques, we implemented a library for efficient type switching in C++. We used the library to rewrite existing code that relied heavily on visitors, and discovered that the resulting code became much shorter, simpler, and easier to maintain and comprehend.

We used the library to rewrite existing code that relied heavily on visitors, and discovered that the resulting code became much shorter, simpler, and easier to maintain and comprehend.

References

[1] clang: a C language family frontend for LLVM. `http://clang.llvm.org/`, 2007.

[2] A. Appel, L. Cardelli, K. Fisher, C. Gunter, R. Harper, X. Leroy, M. Lillibridge, D. B. MacQueen, J. Mitchell, G. Morrisett, J. H. Reppy, J. G. Riecke, Z. Shao, and C. A. Stone. Principles and a preliminary design for ML2000. March 1999. URL `http://flint.cs.yale.edu/flint/publications/ml2000.html`.

[3] Blind Review. Technical report submitted as supplementary material. Technical Report XXXX, University, Nov. 2011.

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

[5] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An experimental applicative language. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, LFP '80, pages 136–143, New York, NY, USA, 1980. ACM. doi: http://doi.acm.org/10.1145/800087.802799. URL `http://doi.acm.org/10.1145/800087.802799`.

[6] W. Burton, E. Meijer, P. Sansom, S. Thompson, and P. Wadler. Views: an extension to haskell pattern matching. 1996.

[7] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. Itanium C++ ABI, March 2001. http://www.codesourcery.com/public/cxx-abi/.

[8] S. Cook, D. Dechev, and P. Pirkelbauer. The IPR Query Language. Technical report, Texas A&M University, Oct. 2004. URL `http://parasol.tamu.edu/pivot/`.

[9] W. R. Cook. Object-oriented programming versus abstract data types. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 151–178, London, UK, 1991. Springer-Verlag. ISBN 3-540-53931-X. URL `http://portal.acm.org/citation.cfm?id=648142.749835`.

[10] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. (The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/ cousot`.).

[11] O.-J. Dahl. *SIMULA 67 common base language, (Norwegian Computing Center. Publication)*. 1968. ISBN B0007JZ9J6.

[12] E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. published as [**?**]WD:EWD472pub, Jan. 1975. URL `http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD472.PDF`.

[13] S. Don, G. Neverov, and J. Margetson. Extensible pattern matching via a lightweight language extension. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 29–40, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: http://doi.acm.org/10.1145/1291151.1291159. URL `http://doi.acm.org/10.1145/1291151.1291159`.

[14] G. Dos Reis and J. Järvi. What is generic programming? In *Proceedings of the First International Workshop of Library-Centric Software Design (LCSD '05). An OOPSLA '05 workshop*, Oct. 2005.

[15] Edison Design Group. C++ Front End, July 2008. http://www.edg.com/.

[16] S. M. (editor). Haskell 2010 language report, July 2010. `http://www.haskell.org/onlinereport/haskell2010/`.

[17] B. Emir. *Object-oriented pattern matching*. PhD thesis, Lausanne, 2007. URL `http://library.epfl.ch/theses/?nr=3899`.

[18] M. D. Ernst, C. S. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98, the 12th European Conference on Object-Oriented Programming*, pages 186–211, Brussels, Belgium, July 20-24, 1998.

[19] D. J. Farber, R. E. Griswold, and I. P. Polonsky. Snobol , a string manipulation language. *J. ACM*, 11:21–30, January 1964. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321203.321207. URL `http://doi.acm.org/10.1145/321203.321207`.

[20] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 406–431, London,

UK, UK, 1993. Springer-Verlag. ISBN 3-540-57120-5. URL http://portal.acm.org/citation.cfm?id=646151.679366.

[21] J. Garrigue. Programming with polymorphic variants. In *ACM Workshop on ML*, Sept. 1998. URL http://www.math.nagoya-u.ac.jp/ garrigue/papers/variants.ps.gz.

[22] M. Gibbs and B. Stroustrup. Fast dynamic casting. *Softw. Pract. Exper.*, 36:139–156, February 2006. ISSN 0038-0644. doi: 10.1002/spe.v36:2. URL http://dl.acm.org/citation.cfm?id=1115606.1115608.

[23] J. Y. Gil and Y. Zibin. Efficient subtyping tests with pq-encoding. *ACM Trans. Program. Lang. Syst.*, 27:819–856, September 2005. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1086642.1086643. URL http://doi.acm.org/10.1145/1086642.1086643.

[24] J. Gimpel. The theory and implementation of pattern matching in snobol4 and other programming languages. Bibliography of Numbered SNOBOL4 Documents, 1971.

[25] N. Glew. Type dispatch for named hierarchical types. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, ICFP '99, pages 172–182, New York, NY, USA, 1999. ACM. ISBN 1-58113-111-9. doi: http://doi.acm.org/10.1145/317636.317797. URL http://doi.acm.org/10.1145/317636.317797.

[26] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language (preliminary report). In D. Suciu and G. Vossen, editors, *International Workshop on the Web and Databases (WebDB)*, May 2000. Reprinted in *The Web and Databases, Selected Papers*, Springer LNCS volume 1997, 2001.

[27] ISO. Working draft, standard for programming language C++. Technical Report N3291=11-0061, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Apr. 2011. URL http://www.open-std.org/JTC1/sc22/wg21/prot/14882fdis/n3291.pdf.

[28] S. P. Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[29] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, London, UK, 1987. Springer-Verlag. ISBN 0-387-17219-X. URL http://portal.acm.org/citation.cfm?id=28220.28222.

[30] S. Krishnamurthi, M. Felleisen, and D. Friedman. Synthesizing object-oriented and functional design to promote re-use. In E. Jul, editor, *ECOOP'98 - Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 91–113. Springer Berlin / Heidelberg, 1998. URL http://dx.doi.org/10.1007/BFb0054088. 10.1007/BFb0054088.

[31] F. Le Fessant and L. Maranget. Optimizing pattern matching. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 26–37, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0. doi: http://doi.acm.org/10.1145/507635.507641. URL http://doi.acm.org/10.1145/507635.507641.

[32] K. Lee, A. LaMarca, and C. Chambers. Hydroj: object-oriented pattern matching for evolvable distributed systems. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 205–223, New York, NY, USA, 2003. ACM. ISBN 1-58113-712-5. doi: http://doi.acm.org/10.1145/949305.949324. URL http://doi.acm.org/10.1145/949305.949324.

[33] A. Leung. Prop: A c++ based pattern matching language. Technical report, Courant Institute of Mathematical Sciences, New York University, 1996. URL http://www.cs.nyu.edu/leunga/prop.html.

[34] B. Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM Press. ISBN 0-89791-266-7. doi: http://doi.acm.org/10.1145/62138.62141.

[35] J. Liu and A. C. Myers. Jmatch: Iterable abstract pattern matching for java. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, PADL '03, pages 110–127, London, UK, UK, 2003. Springer-Verlag. ISBN 3-540-00389-4. URL http://portal.acm.org/citation.cfm?id=645773.668088.

[36] A. Löh and R. Hinze. Open data types and open functions. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '06, pages 133–144, New York, NY, USA, 2006. ACM. ISBN 1-59593-388-3. doi: http://doi.acm.org/10.1145/1140335.1140352. URL http://doi.acm.org/10.1145/1140335.1140352.

[37] Microsoft Research. Phoenix compiler and shared source common language infrastructure. http://research.microsoft.com/phoenix/, 2005.

[38] B. Milewski. Haskell - the pseudocode language for c++ template metaprogramming. In *BoostCon'11*, 2011. URL http://boostcon.boost.org/2011-resources.

[39] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990. ISBN 0262132559.

[40] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 61–76, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-00904-3. URL http://portal.acm.org/citation.cfm?id=1765931.1765938.

[41] M. Odersky and P. Wadler. Pizza into java: Translating theory into practice. In *In Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 146–159. ACM Press, 1997.

[42] 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 (second edition. Technical Report LAMP-REPORT-2006-001, Ecole Polytechnique Federale de Lausanne, 2006.

[43] C. Okasaki. Views for standard ml. In *Workshop on ML*, 1998.

[44] N. Oosterhof. Application patterns in functional languages, 2005. URL http://essay.utwente.nl/57656/.

[45] S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[46] P. Pirkelbauer. *Programming Language Evolution and Source Code Rejuvenation*. PhD thesis, Texas A&M University, December 2010. URL http://repository.tamu.edu/handle/1969.1/ETD-TAMU-2010-12-8894.

[47] T. Ramananandro, G. Dos Reis, and X. Leroy. Formal verification of object layout for c++ multiple inheritance. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 67–80, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: http://doi.acm.org/10.1145/1926385.1926395. URL http://doi.acm.org/10.1145/1926385.1926395.

[48] G. D. Reis and B. Stroustrup. A principled, complete, and efficient representation of c++. In *Proc. Joint Conference of ASCM 2009 and MACIS 2009*, volume 22 of *COE Lecture Notes*, pages 407–421, December 2009.

[49] J. G. Rossie, Jr. and D. P. Friedman. An algebraic semantics of subobjects. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '95, pages 187–199, New York, NY, USA, 1995. ACM. ISBN 0-89791-703-0. doi: http://doi.acm.org/10.1145/217838.217860. URL http://doi.acm.org/10.1145/217838.217860.

[50] S. Ryu, C. Park, and G. L. S. Jr. Adding pattern matching to existing object-oriented languages. In *2010 International Workshop on Foundations of Object-Oriented Languages*, 2010. URL http://ecee.colorado.edu/ siek/FOOL2010/ryu.pdf.

[51] J. L. Schilling. Optimizing away c++ exception handling. *SIGPLAN Not.*, 33:40–47, August 1998. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/286385.286390. URL http://doi.acm.org/10.1145/286385.286390.

[52] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages*

Conference, volume 2789 of LNCS, pages 214–223. Springer-Verlag, August 2003.

[53] D. A. Spuler. Compiler Code Generation for Multiway Branch Statements as a Static Search Problem. Technical Report Technical Report 94/03, James Cook University, Jan. 1994.

[54] B. Stroustrup. A rationale for semantically enhanced library languages. In *LCSD '05*, October 2005.

[55] H. Sutter and J. Hyslop. Polymorphic exceptions. *C/C++ Users Journal*, 23(4), 2005.

[56] Tom Duff. Duff's Device, Aug 1988. http://www.lysator.liu.se/c/duffs-device.html.

[57] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc. ISBN 3-387-15975-4. URL `http://portal.acm.org/citation.cfm?id=5280.5281`.

[58] D. Vandevoorde and N. Josuttis. *C++ templates: the complete guide*. Addison-Wesley, 2003. ISBN 9780201734843. URL `http://books.google.com/books?id=yQU-NlmQb_UC`.

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

[60] T. L. Veldhuizen. C++ templates are Turing complete. `www.osl.iu.edu/ tveldhui/papers/2003/turing.pdf`, 2003.

[61] P. Wadler. The expression problem. Mail to the java-genericity mailing list. URL `http://www.daimi.au.dk/ madst/tool/papers/expression.txt`.

[62] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 307–313, New York, NY, USA, 1987. ACM. ISBN 0-89791-215-2. doi: http://doi.acm.org/10.1145/41625.41653. URL `http://doi.acm.org/10.1145/41625.41653`.

[63] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety prooffor multiple inheritance in c++. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 345–362, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: http://doi.acm.org/10.1145/1167473.1167503. URL `http://doi.acm.org/10.1145/1167473.1167503`.

[64] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 241–252, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0. doi: http://doi.acm.org/10.1145/507635.507665. URL `http://doi.acm.org/10.1145/507635.507665`.

[65] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *Proc. FOOL 12*, Jan. 2005. `http://homepages.inf.ed.ac.uk/wadler/fool`.