Open and Efficient Type Switch for C++

Yuriy Solodkyy   Gabriel Dos Reis   Bjarne Stroustrup

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

Abstract

Selecting operations based on the run-time type of an object is key to many object-oriented and functional programming techniques. We present a technique for implementing open and efficient type switching on hierarchical extensible data types. The technique is general and copes well with C++ multiple inheritance.

To simplify experimentation and gain realistic performance using production-quality compilers and tool chains, we implement a type switch construct as an ISO C++11 library, called *Mach7*[1]. This library-only implementation provides concise notation and outperforms the visitor design pattern, commonly used for case analysis on types in object-oriented programming. For closed sets of types, its performance roughly equals equivalent code in functional languages, such as OCaml and Haskell. The type-switching code is easier to use and is more expressive than hand-coded visitors are. The library is non-intrusive and circumvents most of the extensibility restrictions typical of the visitor design pattern. It was motivated by applications involving large, typed, abstract syntax trees.

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

General Terms   Languages, Design

Keywords   Type Switch, Typecase, Visitor Design Pattern, Memoization, C++

1. Introduction

Classic algebraic data types as seen in functional languages are closed and their variants are disjoint, which allows for ef-

---

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

ficient implementation of case analysis on such types. Data types in object-oriented languages are extensible and hierarchical (implying that variants are not necessarily disjoint). To be general, case analysis on such types must be *open*: allow for independent extensions, modular type-checking, and dynamic linking. To be accepted for production code, its implementation must also equal or outperform all known workarounds. Existing approaches to case analysis on hierarchical extensible data types are either efficient or open, but not both. Truly open approaches rely on expensive class-membership tests combined with decision trees. Efficient approaches rely on sealing either the class hierarchy or the set of functions, which loses extensibility (§2.5). Consider for example a simple expression language:

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

In an object-oriented language without direct support for algebraic data types, the type representing an expression tree in the language will typically be encoded as an abstract base class, with derived classes implementing specific variants:

```
struct Expr { virtual int eval() = 0; };
struct Value : Expr { ⋯ int eval(); int value; };
struct Plus  : Expr { ⋯ Expr& e₁; Expr& e₂; };
```

A simple evaluator for this language can be implemented with the aid of a virtual function eval() declared in the base class Expr. The approach is *intrusive*, however, as the base class has to be modified every time we add a new functionality. With *Mach7*, we offer an external introspection of objects with case analysis:

```
int eval(const Expr& e)
{
  Match(e)
    Case(const Value& x) return x.value;
    Case(const  Plus& x) return eval(x.e₁)+eval(x.e₂);
    Case(const Minus& x) return eval(x.e₁)−eval(x.e₂);
    Case(const Times& x) return eval(x.e₁)∗eval(x.e₂);
    Case(const Divide& x) return eval(x.e₁)/eval(x.e₂);
  EndMatch
}
```

This is all a user of *Mach7* has to write. The syntax is provided without any external tool support or additional definitions from the user. The library is implemented in standard ISO C++11 [28] with template meta-programming, and a few macros (§3.7). It runs about as fast as OCaml and Haskell equivalents (§4.3), and occasionally comes close or outperforms the handcrafted C++ code based on the *visitor design pattern* (§4.1).

The ideas and the *Mach7* library were motivated by our unsatisfactory experiences working with various C++ front-ends and program analysis frameworks [1, 2, 15, 36]. The problem was not in the frameworks per se, but in the fact that we had to use the *visitor design pattern* [21] to inspect, traverse, and elaborate abstract syntax trees of target languages. We found visitors unsuitable to express application logic directly, surprisingly hard to teach students, and often slower than handcrafted workaround techniques. Instead, users were relying on dynamic casts in many places, often nested, thus preferring shorter, cleaner, and more direct code to visitors. The consequential performance penalty was usually not discovered until later, when it was hard to remedy.

1.1 Summary

This paper makes the following contributions:

- A technique for implementing open and efficient type switching on extensible hierarchical data types as seen in object-oriented languages.
- The technique delivers equivalent performance to that of closed algebraic data types, and outperforms the visitor design pattern for open class hierarchies.
- A library implementation provides the notational convenience of functional-language constructs for selecting among types.
- A new approach to type switching that combines subtype tests and type conversion, outperforming approaches that combine subtype tests (even constant-time ones) with decision trees, even over small class hierarchies.
- Complete integration with the existing C++ abstraction mechanisms and object model.
- A constant-time function that partitions a set of objects with the same static type into equivalence classes based on the inheritance path of the static type within the dynamic type.

In particular, our technique for efficient type switching:

- Is open by construction (§3.2), non-intrusive, and avoids the control inversion, as typical for visitors.
- Works in the presence of multiple inheritance, both repeated and virtual, as well as in generic code (§3.7).
- Comes close to and often significantly outperforms the workaround techniques used in practice, e.g. the visitor design pattern, without sacrificing extensibility (§4.1).
- Does not require any changes to the C++ object model or computations at link or load time.

- Can be used in object-oriented languages with object models similar to that of C++.
- Is implemented as a library written in ISO Standard C++11.

This is the first technique for handling type switching efficiently in the presence of the general multiple inheritance present in C++. Being a library, *Mach7* can be used with any ISO C++11 compiler (e.g., Microsoft, GNU, or Clang) without requiring any additional tools or preprocessors. It sets a new threshold for acceptable performance, brevity, clarity, and usefulness of a forthcoming compiler implementation of the open type switch in C++.

2. Overview

A *class hierarchy* is a partially ordered set $(H, <:)$ where H is a potentially open set of classes and $<:$ is a reflexive, transitive and anti-symmetric *subtyping relation* on H. We use *class* and *type* interchangeably. When two classes are in a subtyping relation $D <: B$, the class D is said to be a (possibly indirect) *derived class* (or subtype) of B; the class B is called a (possibly indirect) *base class* (or supertype) of D.

The *dynamic type of an object* is the type used to create it. The *static type of an expression* is the type of that expression as given by the static semantics.

2.1 Type Switch

In general, a *type switch* or *typecase* is a multiway branch statement that distinguishes values based on their type. In a multi-paradigm programming language like C++, which supports parametric, ad-hoc, and subtyping polymorphisms, such a broad definition subsumes numerous different type-case constructs studied in the literature [24, 26, 51]. In this work, we only look at typecasing scenarios based on the class inheritance of C++, similar to those studied by Glew [24]. We use the term *type switch* instead of a broader *typecase* to stress the run-time nature of the type analysis similar to how regular **switch**-statement of C++ performs case analysis of values at run time.

The term *object descriptor* means either a pointer or a reference to an object. Given an object descriptor, called *subject*, of static type S referred to as the *subject type*, and a list of *target types* T_i associated with the branches, a type switch statement needs to identify a suitable clause m based on the dynamic type D <: S of the subject as well as a suitable conversion that coerces the subject to the target type T_m. Due to multiple inheritance, types T_i may not all directly derive from the static type S. However, the type of the applicable clause T_m will necessarily have to be a supertype of the subject's dynamic type D <: T_m. A hypothetical type switch statement, not currently supported by C++, may look as following:

switch (subject) { **case** T_1: s_1; ... **case** T_n: s_n; }

There is no need for an explicit *default clause* in our setting because it is semantically equivalent to a case clause guarded by the subject type: **case** S: s. The only semantic difference such a choice makes is in the treatment of null pointers. One may naively think that null pointers should be handled by the default clause. However, not distinguishing between invalid object and valid object of a known static but unknown dynamic type may lead to nasty run-time errors.

Similar control structures exist in many programming languages, e.g. *match* in Scala [39], *case* in Haskell [29] and ML [37], *typecase* in Modula-3 [6] and CLOS (as a macro), *tagcase* in CLU [34], *union case* in Algol 68, and date back to at least Simula's *Inspect* statement [11]. The statement can, in general, be given numerous plausible semantics:

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

The list is not exhaustive and depending on a language, any of them is a plausible choice. Functional languages, for example, often prefer first-fit semantics because it is similar to case analysis in mathematics. Object-oriented languages are typically inclined to best-fit semantics due to its similarity to overload resolution and run-time dispatch; however, some do opt for first-fit semantics to mimic the functional style: e.g. Scala [39]. Exact-fit semantics can often be seen in languages supporting discriminated union types (sum types): e.g. *variant records* in Pascal, Ada and Modula-2, *oneof* and *variant objects* in CLU, *unions* in C and C++, etc. All-fit and any-fit semantics might be seen in languages based on predicate dispatching [20] or guarded commands [14], where a predicate can be seen as a characteristic function of a type, while logical implication – as subtyping.

2.2 The Expression Problem

Type switching is related to a more general problem manifesting the differences in functional and object-oriented programming styles. Conventional algebraic datatypes, as found in most functional languages, allow for easy addition of new functions on existing data types. However, they fall short in extending data types themselves (e.g. with new constructors), which requires modifying the source code. Object-oriented languages make data type extension trivial through inheritance, but the addition of new functions operating on these classes typically requires changes to the class definition. This dilemma is known as the *expression problem* [10, 52].

Classes differ from algebraic data types in two important ways. Firstly, they are *extensible*, for new variants can be added later by inheriting from the base class. Secondly, they are *hierarchical* and thus typically *non-disjoint* since variants can be inherited from other variants and form a subtyping relation between themselves [24]. In contrast, variants in conventional algebraic data types are *disjoint* and *closed*. Some functional languages e.g. ML2000 [3] and its predecessor, Moby, were experimenting with *hierarchical extensible sum types*, which are closer to object-oriented classes then algebraic data types are, but, interestingly, they provided no facilities for performing case analysis on them.

Zenger and Odersky refined the expression problem in the context of independently extensible solutions [56] 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.

While these requirements were formulated for extensible data types with disjoint variants, object-oriented languages primarily deal with hierarchical data types. We thus found it important to state explicitly an additional requirement based on the Liskov substitution principle [33]:

- *Substitutability*: Operations expressed on more general data variants should be applicable to ones that are more specific.

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 [22, 35] and object-oriented camps [25, 32, 41, 55], but very few have made their way into one of the mainstream languages. We refer the reader to Zenger and Odersky's original manuscript for a discussion of the approaches [56]. Most of the discussed object-oriented solutions focused on the visitor design pattern and its extensions, which even today seem to be the most commonly used approach to dealing with the expression problem in object-oriented languages.

A lot has been written about the visitor design pattern [21, 40, 41, 55]. Its advantages include *extensibility of functions*, *speed*, and *being a library solution*. Nevertheless, the so-

lution is *intrusive*, *specific to hierarchy*, and requires a lot of *boilerplate code* to be written. It also introduces *control inversion*, but, most importantly, – *hinders extensibility* of classes.

2.3 An Open Type Switch

Type switch alone does not solve the expression problem in the context of an object-oriented language, for the existing code may have to be modified to consider new variants. Relying on a default clause is not an acceptable solution in this situation, because often the only reasonable default behavior is to raise an exception. Zenger and Odersky observed that defaults transform type errors that should manifest statically into runtime exceptions [56]. In our experience, newly added variants were more often extending an existing variant than creating an entirely disjoint one. In a compiler, for example, a new kind of type expression will typically extend a TypeExpression variant, while a new form of annotation will extend an Annotation variant, thus not extending the root ASTNode directly. Due to the substitutability requirement, this new variant will be treated as a variant it extends in all the existing code. The functions that will be affected by its addition and thus have to be modified will be limited to functions directly analyzing the variant it extends and not providing a default behavior.

To account for this subtlety of extensible hierarchical data types, we use a term *open type switch* to refer to a type switch that satisfies all the requirements of an *open solution to the expression problem* stated above except for the *no modification or duplication* requirement. We loosen it to allow modification of functions for which the newly added variant becomes a disjoint (orthogonal) case not handled by a default clause. We believe that the loosened requirement allows us to express pragmatically interesting restrictions that developers are willing to live with. Furthermore, open type switch overcomes all the major shortcomings of the visitor design pattern:

- Case analysis with an open type switch is non-intrusive as it inspects the hierarchy externally and can be applied retroactively.
- New variants can be accounted for in the newly written code and will be seen as a base class or default in the existing code.
- The affected functions are limited to those for which the newly added variant is a disjoint case.
- The code avoids the control inversion and the need for boilerplate code that visitors introduce, and is thus a more direct expression of the intent.

2.4 C++ Specifics: Subobjects

C++ supports two kinds of multiple inheritance: *non-virtual* inheritance and *virtual* inheritance [18]. The difference between the two only arises in situations where a class indirectly inherits from the same base class via more than one

path in its class hierarchy. Rigorous accounts of C++ multiple inheritance semantics use the notion of *subobject* [44].
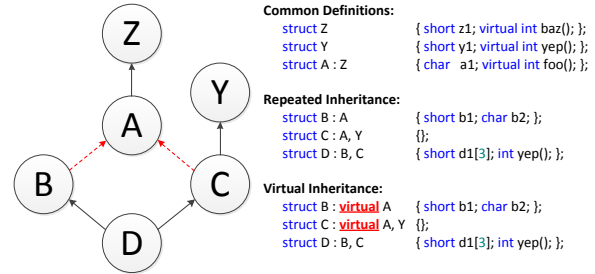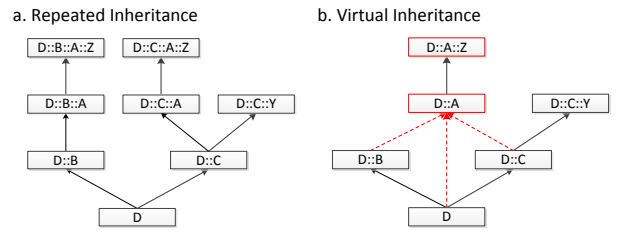

Figure 1. Multiple Inheritance in C++

Consider the simple class hierarchy in Figure 1(1). Class D indirectly inherits from class A through its B and C base classes. In this case, the user may opt to keep distinct subobjects of class A (repeated inheritance) or a shared one (virtual inheritance) by specifying how B and C inherit from A. The kind of inheritance is thus not a property of a given class, but a property of an inheritance relation between classes and it is possible to mix the two.

A class hierarchy gives rise to a *subobject graph*, where a given class node may be replicated when inherited repeatedly or left shared when inherited virtually. The edges in such a graph represent *subobject containment* and indicate whether such containment is shared or exclusive. Every class C in the class hierarchy will have its own subobject graph representing the subobjects of an object of dynamic type C. Figure 1(2) shows subobject graph for class D obtained for the class hierarchy in Figure 1(1) under repeated (a) and virtual (b) inheritance of class A by classes B and C. The shared containment is indicated with the dashed arrows, while exclusive – with the solid ones.

C++'s notion of multiple inheritance is fundamentally about subobjects, not just types. Virtual inheritance is about sharing the base-class subobjects, whereas non-virtual inheritance reflects distinction in base-class subobjects from distinct class inheritance paths [18].

An object descriptor of static type A referencing an object of the dynamic type C can be understood as any $C::*::A$-node in the subobject graph of C. Casts can be understood as a change from one subobject to another. We use the terms *source subobject* and *target subobject* to refer to the argument and result of the cast, respectively. Their static

types will be referred to as *source type* and *target type* respectively. C++ distinguishes three kinds of casts: upcasts, downcasts, and crosscasts.

An *upcast* is a cast from a derived class to one of its bases. When the base class is unambiguous, such casts are implicit and require no additional annotations. When the base class is ambiguous, cast failure is manifested statically in the form of a compile-time error. For example, this is the case with casting D to A under repeated multiple inheritance of A, in which case the user needs to explicitly cast the object to B or C first in order to indicate the desired subobject and resolve the ambiguity. In some cases, however, introduction of such an explicit cast is not possible: e.g. in implicit conversions generated by the compiler to implement covariant return types, crosscasts or conversions in generic code. This does not mean that in such cases we violate the Liskov substitution principle: the classes are still in a subtyping relation, but an implicit conversion is not available.

A *downcast* is a cast from a base class to one of its derived classes. The cast has to determine at run-time whether the source subobject is contained by a subobject of the target type in the dynamic type's subobject graph. Failure of such a cast is manifested dynamically at run-time.

A *crosscast* is a cast between classes that are not necessarily related by inheritance except by sharing a common derived class (subclass). Accordingly to the C++ semantics such cast is defined to be a composition of upcast to target type and downcast to the dynamic type. While the downcast to the dynamic type is always guaranteed to succeed regardless of the source subobject, the upcast to the target type may be ambiguous, in which case the cast will fail at run-time. A cast from Y to B inside an object of dynamic type D in Figure 1(2a,2b) is an example of a successful crosscast. A similar cast from Y to A inside D under the repeated inheritance in Figure 1(2a) will fail because of the ambiguous upcast from D to A.

An interesting artifact of these distinctions can be seen in an example of casting a subobject of type Z to a subobject of type A in Figure 1(2a). The subobject D::B::A::Z will be successfully cast to D::B::A, while the D::C::A::Z will be successfully cast to D::C::A. These casts do not involve downcasting to D followed by an upcast to A, which would be ambiguous, but instead take the dynamic type of a larger subobject (D::B or D::C) that the source subobject is contained in into account in order to resolve the ambiguity. A similar cast from Y to A will fail; should Y have also been non-virtually derived from Z, the cast from D::C::Y::Z to A would have failed. This shows that the distinction between crosscast and downcast is not based solely on the presence of a subtyping relation between the source and target types, but also on the actual position of the source subobject in the dynamic type's subobject graph.

The C++ inheritance model, presented here informally, further complicates the definition and implementation of a type switch compared to simpler models. We have to define the type switch so that only unambiguous casting between a source and a target within an object is possible. That is, the implementation of the cast between source and target subobjects must take into account the location of the source subobject in the subobject graph, rather than just the dynamic and target types, which would suffice for a simple subtype testing. Of course, every use of dynamic casting and every implicit cast are type safe [53].

2.5 Existing Approaches to Type Case Analysis

The closed nature of algebraic data types allows for their efficient implementation. The traditional compilation scheme assigns unique (and often small and sequential) tags to every variant of the algebraic data type and type switching is then simply implemented with a multi-way branch [46] (usually a jump table) over all the tags [4]. Dealing with extensible hierarchical data types makes this approach infeasible:

- *Extensibility* implies that the compiler may not know the exact set of all the derived classes until link-time (due to *separate compilation*) or even run-time (due to *dynamic linking*).
- *Substitutability* implies that we should be able to match tags of derived classes against case labels representing tags of base classes.
- The presence of *multiple inheritance* might require pointer adjustments that are not known at compile time (e.g. due to virtual base classes, ambiguous base classes or crosscasting).

There are two main approaches to implementing case analysis on extensible hierarchical data types discussed in the literature.

The first approach is based on either explicit or implicit sealing of the class hierarchy on which type switching can be performed. C++11, for example, allows the user to prohibit further derivation by specifying a class to be "final" [28], similar to Scala and Java. The compiler then may use the above tag allocation over all variants to implement type analysis [19, §4.3.2]. In some cases, the sealing may happen implicitly. For example, languages with both internal and external linkage may employ the fact that classes with internal linkage will not be externally accessible and are thus effectively sealed. While clearly efficient, the approach is not open as it avoids the question rather than answers it.

The broader problem with this approach is that techniques that rely on unique or sequential compile or link-time constants violate independent extensibility since without a centralized authority there is no guarantee same constant will not be chosen in a type-unsafe manner by independent extensions. Updating such constants at load time may be too costly even when possible.

An important practical solution that follows this approach is the visitor design pattern [21]. The set of visit methods in a visitor's interface essentially seals the class hierarchy.

Extensions have been proposed in the literature [55], but they have problems of their own, as discussed in §5.

The second approach employs type inclusion tests combined with decision trees [5] to avoid unnecessary checks. Its efficiency is then entirely focused on the efficiency of type inclusion tests [7, 9, 12, 17, 23, 31, 45, 50, 54, 57].

C++ has handled general dynamic casting since 1987, when multiple inheritance was added to the language [47]. Wirth later presented a technique that can be used to implement subtype tests by traversing a linked list of types [54]. His encoding required little space, but ran in time proportional to the distance between the two types in the class hierarchy. A trivial constant-time type inclusion test can be implemented with a *binary matrix*, encoding the subtyping relation on the class hierarchy [12]. While efficient in time, it has quadratic space requirements, which makes it expensive for use on large class hierarchies. Cohen proposed the first space-efficient constant-time algorithm, but it can only deal with single inheritance [9]. *Hierarchical encoding* is another constant-time test that maps subtype queries into subset queries on bit-vectors [7, 31]. The approach can handle multiple inheritance, but the space and time required for a subtype test in this encoding increases with the size of the class hierarchy; also, Caseau's approach [7] is limited to class hierarchies that are lattices. Schubert's *relative numbering* [45] encodes each type with an interval $[l, r]$, effectively making type inclusion tests isomorphic to a simple range checking. The encoding is optimal in space and time, but it is limited to single inheritance. *PQ-Encoding* of Zibin and Gil employs PQ-trees to improve further space and time efficiency of the constant-time inclusion testing [57]. While capable of handling type inclusion queries on hierarchies with multiple inheritance, the approach makes the closed world assumption and can be costly for use with dynamic linking because it is not incremental. The approach of Gibbs and Stroustrup [23] employs divisibility of numbers to obtain a constant-time type inclusion test. The approach can handle multiple inheritance and was the first constant-time technique to addresses the problem of casts between subobjects. Unfortunately, the approach limits the size of the class hierarchies that can be encoded with this technique. Ducournau proposed a constant-time inclusion test based on the fact that, in an open solution, a class has a known number of base classes, and thus perfect hashes can be used to map them to this-pointer offsets typically used to implement subobject casts [17]. Unfortunately, the approach addresses only virtual multiple inheritance and (similarly to other approaches) relies on load-time computations. An excellent introduction to and detailed analysis of existing constant-time type inclusion tests can be found in [50, 57].

With the exception of work by Gibbs and Stroustrup [23], all the approaches to efficient type-inclusion testing we found in the literature were based on the assumption that *the outcome of a subtyping test as well as the subsequent* *cast depend only on the target type and the dynamic type of the object*. Although that assumption is sound for subtyping tests and subtype casts for shared inheritance (including single), it does not reflect the relationship between subobjects in the general case of multiple inheritance as found in C++.

2.6 The Source of Inefficiency

While constant-time type inclusion tests are invaluable in optimizing subtype tests in programming languages, their use in implementing a type switch is inferior to some workaround techniques. This may prevent wide adoption of a language implementation of such a feature due to its inferior performance. We implemented 3 constant-time type inclusion tests: binary matrix [50], Cohen's algorithm [9], and fast dynamic cast [23] and combined them with a decision tree to implement a type switch on a class hierarchy ideally suited for such scenarios: a perfect binary tree with classes number $2i$ and $2i + 1$ derived from a class number i. Our workaround techniques included the visitor design pattern and a switch on the sealed sequential set of tags.
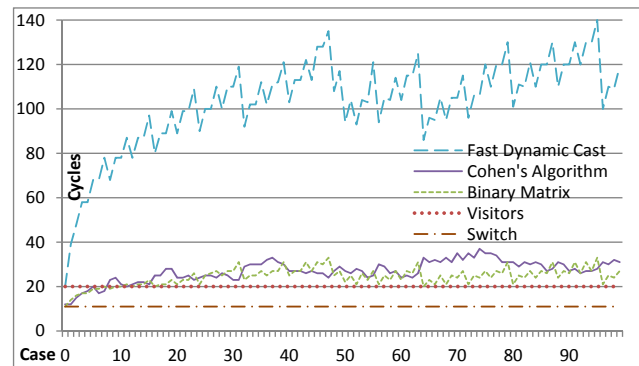

Figure 2. Type switch based on constant-time subtype tests

The chart in Figure 2 shows the number of cycles (Y-axis) each technique took to recognize an object of the dynamic type i (X-axis). Despite known limitations, binary matrix and Cohen's algorithm are some of the fastest known type inclusion tests for single inheritance [50]. It is nonetheless easy to see that the logarithmic cost associated with the decision tree very quickly surpasses the constant overhead of double dispatch (20 cycles) present in the visitor design pattern or the jump-table implementation of the switch on all tags (11 cycles). We expect the cost of techniques capable of handling multiple inheritance to be even higher, especially those addressing casting between subobjects (e.g. fast dynamic cast). The edgy shape of timing results reflects the shape of the class hierarchy used for this experiment.

3. Type Switching

Mach7 explicitly supports at least two encodings of algebraic datatypes: runtime type information discriminant, and numerical tag data member shared by all classes in a given hierarchy. The library handles them differently to let the user

choose between openness and efficiency. The type switch for tagged encoding (§3.1) is simpler and more efficient for many typical use cases, however, making it open eradicates its performance advantages (§4.2).

3.1 An Attractive Non-Solution

While Wirth' linked list encoding was considered slow for subtype testing, it can be adopted for quite efficient type switching on a class hierarchy with no repeated inheritance. The idea is to combine fast switching on closed algebraic datatypes with a loop that tries the tags of base classes when switching on derived tags fails.

For simplicity of presentation we assume a pointer to an array of tags be available directly through the subject's taglist data member. The array is of variable size: its first element is always the tag of the subject's dynamic type, while its end is marked with a dedicated end_of_list marker, distinct from all the tags. The tags in between are topologically sorted according to the subtyping relation with incomparable siblings listed in *local precedence order* – the order of the direct base classes used in the class definition. The list resembles the *class precedence list* of object-oriented descendants of Lisp (e.g. Dylan, Flavors, LOOPS, and CLOS) used there for *linearization* of class hierarchies. We also assume the tag-constant associated with a class D_i is accessible through a static member D_i::class_tag. These simplifications are not essential and the library does not rely on any of them.

A type switch, below, 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 list and jumps back for a rematch:

```
   size_t attempt = 0;
   size_t tag = subject→ taglist[attempt];
ReMatch:
   switch (tag) {
   default:
      tag = subject→ taglist[++attempt];
      goto ReMatch;
   case end_of_list:
      break;
   case D₁::class_tag:
      D₁& match = static_cast⟨D₁&⟩(*subject); s₁;
      break;
      ...
   case Dₙ::class_tag:
      Dₙ& match = static_cast⟨Dₙ&⟩(*subject); sₙ;
      break;
   }
```

The above structure, which we call a *tag switch*, implements a variation of best-fit semantics based on local precedence order. It lets us dispatch to the case clause of the most-specialized class with an overhead of initializing two local variables, compared to an efficient switch used on algebraic data types. Dispatching to a case clause of a base class will take time roughly proportional to the distance between the matched base class and the derived class in the inheritance graph, thus the technique is not constant. When none of the base class tags was matched, we will necessarily reach the end_of_list marker and exit the loop. The default clause, again, can be implemented with a case clause on the subject type's tag: **case** S::class_tag:

The efficiency of the above code crucially depends on the set of tags being small and sequential to justify the use of a jump table instead of a decision tree to implement the switch. This is usually not a problem in closed hierarchies based on tag encoding since the designer of the hierarchy handpicks the tags herself. The use of a static cast however, essentially limits the use of this mechanism to non-repeated inheritance only. This only refers to the way target classes inherit from the subject type – they can freely inherit from other classes. Due to these restrictions, the technique is not open because it may violate independent extensibility. We discuss in §4.2 that making the technique more open will also eradicate its performance advantages.

3.2 An Open but Inefficient Solution

Instead of starting with an efficient solution and trying to make it open, we start with an open solution and try to make it efficient. The following cascading-if statement implements the first-fit semantics for our type switch in a truly open fashion:

if $(T_1*$ match=**dynamic_cast**$\langle T_1*\rangle$(subject)) $\{s_1;\}$ **else**
if $(T_2*$ match=**dynamic_cast**$\langle T_2*\rangle$(subject)) $\{s_2;\}$ **else**
...
if $(T_n*$ match=**dynamic_cast**$\langle T_n*\rangle$(subject)) $\{s_n;\}$

Its main drawback is performance: a typical implementation of **dynamic_cast** takes time proportional to the distance between base and derived classes in the inheritance tree. What is worse is that due to the sequential order of tests, the time to uncover the type in the ith case clause will be proportional to i, while failure to match will take the longest. In a test involving a flat hierarchy of 100 variants, it took 93 cycles to discover the first type and 22760 to discover the last (with their linear combination for the types in between).

Relying on **dynamic_cast** also makes an implicit semantic choice where we are no longer looking for the first/best-fitting type that is in subtyping relation, but for the first/best-fitting type to which a cast is possible from the source sub-object (§2.4).

3.3 A 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 are *functional* (i.e. do not involve any side effects), the next time we execute the switch with the same value x, the same predicate will be enabled first. We thus would like to avoid evaluating preceding predicates and jump to the statement it guards. In a way, we would like the switch to memoize the case enabled for a given 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 preceding it after the target must re-establish any invariant guaranteed by sequential execution (§3.7).

Described code can be easily produced in a compiler setting, but generating it in a library is a challenge. Inspired by Duff's Device [48], we devised a construct that we call *Memoization Device* doing just that in standard C++:

```
typedef decltype(x) T; // T is the type of subject x
static std::unordered_map⟨T,size_t⟩ jump_targets;

switch (size_t& jump_to = jump_targets[x]) {
default: // entered when we have not seen x yet
    if (P₁(x)) { jump_to = 1; case 1: s₁;} else
    if (P₂(x)) { jump_to = 2; case 2: s₂;} else
        ...
    if (Pₙ(x)) { jump_to = n; case n: sₙ;} else
                    jump_to = n + 1;
case n + 1: // none of the predicates is true on x
}
```

The static jump_targets 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 allocate a new entry with its associated data default initialized (to 0 for size_t). 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 jump_to effectively establish association between value x and corresponding predicate, since jump_to is a reference to jump_targets[x]. The last assignment records absence of enabled predicates for x.

To change the first-fit semantics of the above construct into *sequential all-fit*, we remove the **else**-statements and rely on fall-through behavior of the switch. We make the assignments conditional to make sure, only the first is recorded:

```
if (Pᵢ(x)) { if (jump_to ==0) jump_to = i; case i: sᵢ;}
```

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, but keeping the default clause generates faster code. A more important 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. 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 [20].

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. $\forall i \forall x \in \bigcap_j \mathrm{Domain}(P_j).P_{2i+1}(x) \rightarrow P_i(x) \wedge P_{2i}(x) \rightarrow P_i(x) \wedge \neg(P_{2i+1}(x) \wedge P_{2i}(x))$ holds. Examples of such predicates are class membership tests where the truth of testing membership in a derived class implies the truth of testing membership in its base class.

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 (size_t& jump_to = jump_targets[x]) {
default:
    if (P₁(x)) {
        if (P₂(x)) {
            if (P₄(x)) { jump_to = 4; case 4: s₄;} else
            if (P₅(x)) { jump_to = 5; case 5: s₅;}
            jump_to = 2; case 2: s₂;
        } else
        if (P₃(x)) {
            if (P₆(x)) { jump_to = 6; case 6: s₆;} else
            if (P₇(x)) { jump_to = 7; case 7: s₇;}
            jump_to = 3; case 3: s₃;
        }
        jump_to = 1; case 1: s₁;
    } else { jump_to = 0; case 0: ; }
}
```

Our library solution prefers the simpler cascading-if approach only because the necessary code structure can be laid out 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.

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 that do not change the outcome of the predicates. The map can then associate the equivalence class of a value with a target instead of associating the value with it.

In application to type switching, the idea is to use the memoization device to learn the outcomes of type inclusion tests (with **dynamic_cast** used as a predicate). The objects can be grouped into equivalence classes based on their dynamic type: the outcome of each type inclusion test will be the same on all the objects of the same dynamic type. We can use the address of a class' **type_info** object obtained in constant time with the **typeid**() operator as a unique identifier of each dynamic type.

This could have been a solution if we were only interested in class membership. More often than not, however, we will be interested in obtaining a reference to the target type of the subject, and we saw in §2.4 that the cast between the source and target subobjects depends on the position of the source subobject in the dynamic type's subobject graph. We thus would like to have different equivalence classes for different subobjects.

3.4 Virtual Table Pointers

Figure 3 shows a typical object layout generated by a C++ compiler for class D from Figure 1(1) under repeated (1) and virtual (2) inheritance of A. The layouts represent an encoding of the corresponding subobject graphs from Figures 1(2a) and 1(2b) respectively.
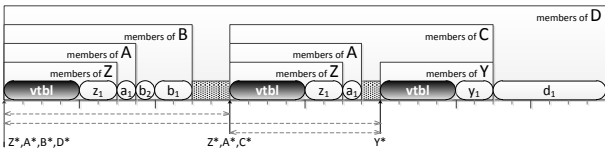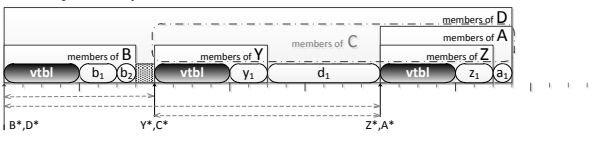

Figure 3. Object Layout under Multiple Inheritance

Due to the extensibility of classes, the layout decisions for classes must be made independently of their derived classes – a property of the C++ object model that we will refer to as *layout independence*. In turn, the layout of derived classes must conform to the layout of their base classes relatively to the offset of the base class within the derived one. For example, the layout of A in C is exactly the same as the layout of A in B and is simply the layout of A. Base classes inherited virtually do not contribute to the fixed layout because they are looked up indirectly at run-time; however, they are not exempt from layout independence, since their lookup rules are agnostic of the concrete dynamic type.

Under non-virtual inheritance, members of the base class are typically laid out before the members of derived class, resulting in the base class being at the same offset as the derived class itself. In our example, the offset of A in B under regular (non-virtual) inheritance of A is 0. Under multiple inheritance, different base classes might be at different offsets in the derived class, which is why pointers of a given static type may be pointing only to certain subobjects in it. These positions are marked in the picture with vertical arrows decorated with the set of pointer types whose values may point into that position. Run-time conversions between such pointers represent casts between subobjects of the same dynamic type and may require adjustments to this-pointer (shown with dashed arrows) for type safety.

A class that declares or inherits a virtual function is called a *polymorphic class*. The C++ standard [28] does not prescribe any specific implementation technique for virtual function dispatch. However, in practice, all C++ compilers use a strategy based on so-called virtual function tables (or vtables for short) for efficient dispatch. The vtable is part of the reification of a polymorphic class type. C++ compilers embed a pointer to a vtable (vtbl-pointer for short) in every object of polymorphic class type (and thus every subobject of that type inside other classes due to layout independence). CFront, the first C++ compiler, puts the vtbl-pointer at the end of an object. The so-called "common vendor C++ ABI" [8] requires the vtbl-pointer to be at offset 0 of an object. We do not have access to the unpublished Microsoft ABI, but we have experimental evidence that their C++ compiler also puts the vtbl-pointer at the start of an object.

While the exact offset of the vtbl-pointer within the (sub)object is not important for this discussion, because of layout independence every (sub)object of a polymorphic type S will have a vtbl-pointer at a predefined offset. Such offset may be different for different static types S, in which case the compiler will know at which offset in type S the vtbl-pointer is located, but it will be the same within any subobject of a static type S. For a library implementation we assume the presence of a function **template** 〈**typename** S〉intptr_t vtbl(**const** S∗s); that returns the address of the virtual table corresponding to the subobject pointed to by s. Such a function can be trivially implemented for the common vendor C++ ABI, where the vtbl-pointer is always at offset 0:

```
template 〈typename S〉 std::intptr_t vtbl(const S* s) {
    static_assert(std::is_polymorphic〈S〉::value, "error");
    return *reinterpret_cast〈const std::intptr_t*〉(s);
}
```

Each of the vtbl fields shown in Figure 3 holds a vtbl-pointer referencing a group of virtual methods known in the object's static type. Figure 4(1) shows a typical layout of virtual function tables together with objects it points to for classes B and D.

Entries in the vtable to the right of the address pointed to by a vtbl-pointer represent pointers to functions, while entries to the left of it represent various additional fields like a pointer to a class' type information, offset to top, offsets

1. Vtable layout with Run-Time Type Information
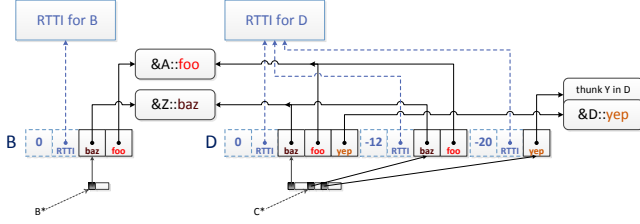
2. Vtable layout without Run-Time Type Information
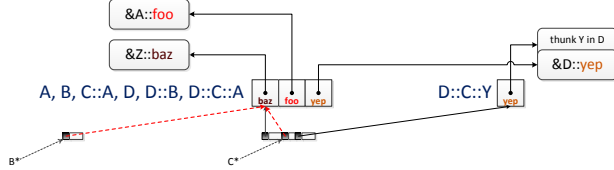

Figure 4. VTable layout with and without RTTI

to virtual base classes, etc. In many implementations, this-pointer adjustments required to dispatch properly the call were stored in the vtable along with function pointers. Today most implementations prefer to use *thunks* or *trampolines* – additional entry points to a function, that adjust this-pointer before transferring the control to the function, – which was shown to be more efficient [16]. Thunks in general may only be needed when virtual function is overridden. In such cases, the overridden function may be called via a pointer to a base class or a pointer to a derived class, which may not be at the same offset in the actual object.

The intuition behind our proposal is to use the values of vtbl-pointers stored inside the object to uniquely identify the subobject in it. There are several problems with the approach, however. First, the same vtbl-pointer is usually shared by multiple subobjects when one of them contains the other. For example, the first vtbl-pointer in Figure 3(1) will be shared by objects of static type Z∗, A∗, B∗ and D∗. This is not a problem for our purpose, because the subobjects of these types will be at the same offset in the object. Secondly, and more importantly, however, there are legitimate optimizations that let the compiler share the same vtable among multiple subobjects of often-unrelated types.

Generation of the *Run-Time Type Information* (or RTTI for short) can typically be disabled with a compiler switch and the Figure 4(2) shows the same vtable layouts once RTTI has been disabled. Since neither baz nor foo were overridden, the prefix of the vtable for the C subobject in D is exactly the same as the vtable for its B subobject, the A subobject of C, or the entire vtable of A and B classes. Such a layout, for example, is produced by Microsoft Visual C++ 11 when the command-line option /GR− is specified. The Visual C++ compiler has been known to unify code identical on binary level, which in some cases may result in sharing of the same vtable between unrelated classes (e.g. when virtual functions are empty).

We now would like to show more formally that in the presence of RTTI, a common vendor C++ ABI compliant implementation would always have all the vtbl-pointers different. To do so, we need a closer look at the notion of subobject, which has been formalized before [43, 44, 53]. We follow here the presentation of Ramamanandro et al [43].

3.5 Subobjects

We assume a program $\mathfrak{P}$ is represented by its class table, which can be queried for inheritance relations between classes. All subsequent definitions are implicitly parameterized over a given program $\mathfrak{P}$. A class B is a *direct repeated base class* of D if B is mentioned in the list of base classes of D without the **virtual** keyword ($D \prec_R B$). Similarly, a class B is a *direct shared base class* of D if B is mentioned in the list of base classes of D with the **virtual** keyword ($D \prec_S B$). A reflexive transitive closure of these relationships $\preceq^* = (\prec_R \cup \prec_S)^*$ defines the *subtyping* relation on types of program $\mathfrak{P}$. A base class *subobject* of a given *complete object* is represented by a pair $\sigma = (h, l)$ with $h \in \{\mathsf{Repeated}, \mathsf{Shared}\}$ representing the kind of inheritance (single inheritance is Repeated with one base class) and l representing the path in a non-virtual inheritance graph. A judgment of the form $\mathfrak{P} \vdash C \prec \sigma \succ A$ states that in a program $\mathfrak{P}$, σ designates a subobject of static type A within an object of type C. Omitting the context $\mathfrak{P}$:

$$\frac{C \prec_S B \qquad B \prec (h,l) \succ A}{C \prec (\mathsf{Shared}, l) \succ A} \qquad C \prec (\mathsf{Repeated}, C :: \epsilon) \succ C$$

$$\frac{C \prec_R B \qquad B \prec (\mathsf{Repeated}, l) \succ A}{C \prec (\mathsf{Repeated}, C :: l) \succ A}$$

ϵ indicates an empty path, but we will generally omit it in writing when understood from the context. In the case of repeated inheritance in Figure 1(1), an object of the dynamic class D will have the following Repeated subobjects: D::C::Y, D::B::A::Z, D::C::A::Z, D::B::A, D::C::A, D::B, D::C, D. Similarly, in case of virtual inheritance in the same example, an object of the dynamic class D will have the following Repeated subobjects: D::C::Y, D::B, D::C, D as well as the following Shared subobjects: D::A::Z, D::Z, D::A. See Figure 1 for illustration.

It is easy to show by structural induction on the above definition, that $C \prec \sigma \succ A \implies \sigma = (h, C :: l_1) \wedge \sigma = (h, l_2 :: A :: \epsilon)$, which simply means that any path to a subobject of static type A within the object of dynamic type C starts with C and ends with A. This observation shows that $\sigma_\perp = (\mathsf{Shared}, \epsilon)$ does not represent a valid subobject. If $\Sigma_\mathfrak{P}$ is the domain of all subobjects in the program $\mathfrak{P}$ extended with $\sigma_\perp$, then a *cast* operation can be understood as a function $\delta : \Sigma_\mathfrak{P} \to \Sigma_\mathfrak{P}$. We use $\sigma_\perp$ to indicate an impossibility of a cast. The fact that δ is defined on subobjects as opposed to actual run-time values reflects the non-coercive nature of

the operation, i.e. the underlying value remains the same. Any implementation of such a function must thus satisfy the following condition:

$$C \prec \sigma_1 \succ A \wedge \delta(\sigma_1) = \sigma_2 \implies C \prec \sigma_2 \succ B$$

i.e. the dynamic type of the value does not change during casting, only the way we reference it does. Following the definitions from §2.4, A is the *source type* and σ_1 is the *source subobject* of the cast, while B is the *target type* and σ_2 is the *target subobject* of it. The type C is the dynamic type of the value being casted. The C++ semantics states more requirements to the implementation of δ: e.g. $\delta(\sigma_\perp) = \sigma_\perp$ etc. but their precise modeling is out of scope of this discussion. We would only like to point out here that since the result of the cast does not depend on the actual value and only on the source subobject and the target type, we can memoize the outcome of a cast on one instance in order to apply its results to another.

3.6 Uniqueness of vtbl-pointers under common ABI

Given a reference a to polymorphic type A that points to a subobject σ of the dynamic type C (i.e. $C \prec \sigma \succ A$ is true), we will use the traditional field access notation a.vtbl to refer to the virtual table of that subobject. The exact structure of the virtual table as mandated by the common vendor C++ ABI is immaterial for this discussion, but we mention a few fields that are important for the reasoning [8, §2.5.2]:

- rtti(a.vtbl): the *typeinfo pointer* points to the typeinfo object used for RTTI. It is always present and is shown as the first field to the left of any vtbl-pointer in Figure 4(1).
- off2top(a.vtbl): the *offset to top* holds the displacement to the top of the object from the location within the object of the vtbl-pointer that addresses this virtual table. It is always present and is shown as the second field to the left of any vtbl-pointer in Figure 4(1). The numeric value shown indicates the actual offset based on the object layout from Figure 3(1).
- vbase(a.vtbl): *Virtual Base (vbase) offsets* are used to access the virtual bases of an object. Such an entry is required for each virtual base class. None is shown in our example in Figure 4(1) since it discusses repeated inheritance, but they will occupy further entries to the left of the vtbl-pointer, when present.

We also use the notation $offset(\sigma)$ to refer to the offset of a given subobject σ within C, known by the compiler.

Theorem 1. *In an object layout that adheres to the common vendor C++ ABI with RTTI enabled, equality of vtbl-pointers of two objects of the same static type implies that they both belong to subobjects with the same inheritance path in the same dynamic class.*
$\forall a_1, a_2 : A \mid a_1 \in C_1 \prec \sigma_1 \succ A \wedge a_2 \in C_2 \prec \sigma_2 \succ A$
$a_1.vtbl = a_2.vtbl \Rightarrow C_1 = C_2 \wedge \sigma_1 = \sigma_2$

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

Let us assume now that $a_1.vtbl = a_2.vtbl$ but $\sigma_1 \neq \sigma_2$. Let $\sigma_1 = (h_1, l_1), \sigma_2 = (h_2, l_2)$

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 base, $\mathsf{vbase}(a_1.vtbl)$ has to be defined inside the vtable according to the ABI, while $\mathsf{vbase}(a_2.vtbl)$ – should not. This would contradict again that both $vtbl$ refer to the same virtual table.

We thus have $h_1 = h_2 = h$. If $h = \mathsf{Shared}$ then there is only one path to such A in C, which would contradict $\sigma_1 \neq \sigma_2$. If $h = \mathsf{Repeated}$ then we must have that $l_1 \neq l_2$. In this case let k be the first position in which they differ: $\forall j < k.l_1^j = l_2^j \wedge l_1^k \neq l_2^k$. 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 $offset(\sigma_1) = \mathsf{off2top}(a_1.vtbl) = \mathsf{off2top}(a_2.vtbl) = offset(\sigma_2)$ since $a_1.vtbl = a_2.vtbl$, which contradicts that the offsets are different. $\square$

Conjecture in the other direction is not true in general as there may be duplicate vtables for the same type present at run-time. This happens in many C++ implementations in the presence of *Dynamically Linked Libraries* (or DLLs for short) as the same class compiled into executable and DLL it loads may have identical vtables 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 static types is not true in general because a derived class can share a vtbl-pointer with its primary base class. The theorem can be reformulated, however, stating that one subobject will necessarily have to contain the other, but that would require bringing in the formalism for subobject containment [53]. The current formulation is sufficient for our purposes.

During construction and deconstruction of an object, the value of a given vtbl-pointer may change. In particular, that value will reflect the fact that the dynamic type of the object is the type of its fully constructed part only. 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.

3.7 Vtable Pointer Memoization

The C++ standard implies that information about types is available at run time for three distinct purposes [8, §2.9.1]:

- to support the **typeid** operator,
- to match an exception handler with a thrown object, and
- to implement the **dynamic_cast** operator.

and if any of these facilities are used in a program that was compiled with RTTI disabled, the compiler shall emit a warning. Some compilers (e.g. Visual C++) additionally let a library check presence of RTTI through a predefined macro, thus letting it report an error if its dependence on RTTI cannot be satisfied. Since our solution depends on **dynamic_cast**, according to the third requirement we implicitly rely on the presence of RTTI and thus fall into the setting that guarantees the preconditions of Theorem 1. Besides, all the objects that will be coming through a particular type switch will have the same static type, and thus the theorem guarantees that different vtbl-pointers will correspond to different subobjects. The idea is thus to group them according to the value of their vtbl-pointer and associate both jump target and the required offset through the memoization device:

```
typedef pair⟨ptrdiff_t,size_t⟩ target_info; //(offset,target)
static unordered_map⟨intptr_t, target_info⟩ jump_targets;
    auto* sptr = &x; // name to access subject
const void* tptr;
target_info& info = jump_targets[vtbl(sptr)];
switch (info.second) {{ default:
```

The code for the ith case now evaluates the required offset on the first entry and associates it and the target with the vtbl-pointer of the subject. The call to adjust_ptr⟨T_i⟩ re-establishes the invariant that match is a reference to type T_i of the subject x.

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

Class std::unordered_map provides amortized constant time access on average and linear in the number of elements in the worst case. We show in the next section that most of the time we will be bypassing traditional access to its elements. We need this extra optimization because, as-is, the type switch is still about 50% slower than the visitor design pattern.

Looking back at the example from §1 and allowing for a few unimportant omissions, the first code snippet corresponds to what the macro *Match*(x) is expanded to when given a subject expression x. In order to see what *Case*(T_i) is expanded to, the second snippet has to be split on the line containing s_i; (excluding s_i; itself, which comes from source) and the second part (i.e. } here) moved in front of the first one. The macro thus closes the scope of the previous case clause before starting the new one. *Case*'s expansion only relies on names introduced by *Match*(x), its argument T_i, and a constant i, which can be generated from the __LINE__ macro, or, better yet, the __COUNTER__ macro when supported by the compiler. The *EndMatch* macro simply closes the scopes (i.e. }} here). We refer the reader to the library source code for further details.

3.8 Minimization of Conflicts

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

Knowing that vtbl-pointers point into an array of function pointers, we should expect them to be aligned accordingly and thus have a few lowest bits as zero. Moreover, since many derived classes do not introduce new virtual functions, the size of their virtual tables remains the same. When allocated sequentially in memory, we can expect a certain number of lowest bits in the vtbl-pointers pointing to them to be the same. These assumptions, supported by actual observations, made virtual table pointers of classes related by inheritance ideally suitable for hashing: the values obtained by throwing away the common bits on the right were compactly distributed in small disjoint ranges (§4.4). We use them to address a cache built on top of the hash table in order to eliminate a hash table lookup in most of the cases.

Let Ξ be the domain of integral representations of pointers. Given a cache with 2^k entries, we use a family of hash functions $H_{kl} : \Xi \to [0..2^k - 1]$ defined as $H_{kl}(v) = v/2^l \mod 2^k$ to index the cache, where $l \in [0..32]$ (assuming 32 bit addresses) is a parameter modeling the number of common bits on the right. Division and modulo are implemented with bit operations since the denominator in each case is a power of 2, which in turn explains the choice of the cache size.

Given a hash function H_{kl}, pointers v' and v'' are said to be in *conflict* when $H_{kl}(v') = H_{kl}(v'')$. For a given set of pointers $V \in 2^\Xi$, we can always find such k and l that H_{kl} will render no conflicts between its elements, but the required cache size 2^k can be too large to justify the use of memory. The value K such that $2^{K-1} < |V| \le 2^K$ is the smallest value of k under which absence of conflicts is still possible. We thus allow k to vary only in the range $[K, K + 1]$ to ensure that the cache size is never more than 4 times bigger than the minimum required cache size.

Given a set $V = \{v_1, ..., v_n\}$, we would like to find a pair of parameters (k, l) such that H_{kl} will render the least

number of conflicts on the elements of V. Since for a fixed set V, parameters k and l vary in a finite range, we can always find the optimal (k, l) by trying all the combinations. Let $H_{kl}^V : V \to [0..2^k-1]$ be the hash function corresponding to such optimal (k, l) for the set V.

In our setting, the set V represents the set of vtbl-pointers coming through a particular type switch. While the exact values of these pointers are not known until run-time, their offsets from the module's base address are. This is generally sufficient to estimate optimal k and l in a compiler setting. In the library setting, we recompute them after a given number of actual collisions in cache.

When H_{kl}^V is injective (renders 0 conflicts on V), the frequency of any given vtbl-pointer v_i coming through the type switch does not affect the overall performance of the switch. However when H_{kl}^V is not injective, we would prefer the conflict to happen on less frequent vtbl-pointers. Given a probability $p(v_i)$ of each vtbl-pointer $v_i \in V$ we can compute the probability of conflict rendered by a given H_{kl}:

$$p_{kl}(V) = \sum_{j=0}^{2^k-1} \left(\sum_{v_i \in V_{kl}^j} p(v_i) \right) \left(1 - \frac{\sum_{v_i \in V_{kl}^j} p(v_i)^2}{\left(\sum_{v_i \in V_{kl}^j} p(v_i) \right)^2} \right)$$

where $V_{kl}^j = \{ v \in V | H_{kl}(v) = j \}$. In this case, the optimal hash function H_{kl}^V can similarly be defined as H_{kl} that minimizes the above probability of conflict on V.

The probabilities $p(v_i)$ can be estimated in a compiler setting through profiling, while in a library setting we let the user enable tracing of frequencies of each vtbl-pointer. This introduces an overhead of an increment into the critical path of execution, and according to our tests degrades the performance by 1-2%. This should not be a problem as long as the overall performance gains from a smaller probability of conflicts happening at run time. Unfortunately, in our tests the significant drop in the number of actual collisions was not reflected in a noticeable decrease in execution time, which is why we do not enable frequency tracing by default. As we will see in §4.4, this was because the hash function H_{kl}^V renders no conflicts on vtbl-pointers in most cases and the few collisions we were getting before inferring the optimal k and l even in non-frequency-based caching where incomparably smaller than the number of successful cache hits.

Assuming uniform distribution of v_i in V and substituting the probability $p(v_i) = \frac{1}{n}$, where $n = |V|$, into the above formula we get:

$$p_{kl}(V) = \sum_{j=0}^{2^k-1} [|V_{kl}^j| \neq 0] \frac{|V_{kl}^j| - 1}{n}$$

We use the Iverson bracket $[\pi]$ here to refer to the outcome of a predicate π as numbers 0 or 1. The value $|V_{kl}^j|$ represents the number of vtbl-pointers $v_i \in V$ that are mapped to the same location j in cache with H_{kl}^V. Only one such vtbl-pointer will actually be present in that cache location at any given time, which is why the value $|V_{kl}^j| - 1$ represents the number of "extra" pointers mapped into the entry j on which a collision will happen. The overall probability of conflict thus only depends on the total number of these "extra" or conflicting vtbl-pointers. The H_{kl}^V obtained by minimization of probability of conflict under uniform distribution of v_i in V is thus the same as the original H_{kl}^V that was minimizing the number of conflicts. An important observation here is that since the exact location of these "extra" vtbl-pointers is not important and only the total number m is, the probability of conflict under uniform distribution of v_i in V is always going to be of the discrete form $\frac{m}{n}$, where $0 \leq m < n$.

4. Evaluation

We performed several independent studies of our approach to demonstrate its effectiveness. The first study compares our approach to the visitor design pattern and shows that the type switch is comparable or faster (§4.1). While we do not advocate for the closed solution of §3.1, we included the comparison of type switching solutions made under open and closed world assumptions (§4.2). Our library supports both solutions with the same surface syntax, which is why we believe many users will try them both before settling on one. The second study does a similar comparison with built-in facilities of Haskell and OCaml and shows that the open type switch for extensible and hierarchical data types can be almost as efficient as its equivalent for closed algebraic data types (§4.3). In the third study we looked at how well our caching mechanisms deal with some large real-world class hierarchies in order to demonstrate that our performance numbers were not established in overly idealistic conditions (§4.4). In the last study we rewrote an existing visitor-based application using our approach in order to compare the ease of use, readability and maintainability of each approach, as well as to show the memory usage and the startup costs associated with our approach in a real application (§4.5).

4.1 Comparison with Visitor Design Pattern

Our comparison methodology involves several benchmarks representing various uses of objects inspected with either visitors or type switching.

The *repetitive* benchmark (REP) performs calls on different objects of the same dynamic 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 twice faster (20 vs. 53 cycles) in this scenario compared to others due to hardware cache and call target prediction mechanisms.

The *sequential* benchmark (SEQ) 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, which is typical of lookup tables, where each entry is implemented with a different derived class.

The *random* benchmark (RND) is the most representative as it randomly makes calls on different objects – probably be the most common usage scenario in the real world.

Presence of *forwarding* in any of these benchmarks 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 switch corresponds to the use of base classes in the case clauses.

The class hierarchy for non-forwarding test was a flat hierarchy of 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.

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

- *Lnx*: 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; x86 binaries
- *Win*: Sony VAIO® laptop with Intel® Core™i5 460M CPU at 2.53 GHz; 6GB of RAM; Windows 7 Pro.
  - G++ 4.6.1 / MinGW executed with -O2; x86 binaries
  - MS Visual C++ 2010 Professional x86/x64 binaries with and without Profile-Guided Optimizations

To improve accuracy, timing in all the configurations was performed with the help of RDTSC instruction available on x86 processors. For every number reported here we ran 101 experiments timing 1,000,000 dispatches each (all through either visitors or type switch). The first experiment was serving as a warm-up, during which the optimal caching parameters were inferred, and typically resulted in an outlier with the largest time. Averaged over 1,000,000 dispatches, the number of cycles per dispatch in each of the 101 experiments was sorted and the median was chosen. We preferred median to average to diminish the influence of other applications and OS interrupts as well as to improve reproducibility of timings between the runs of application. In particular, in the diagnostic boot of Windows, where the minimum of drivers and applications are loaded, we were getting the same number of cycles per iteration 70-80 out of 101 times. Timings in non-diagnostic boots had somewhat larger absolute values, however the relative performance of type switch against visitors remained unchanged and equally well reproducible.

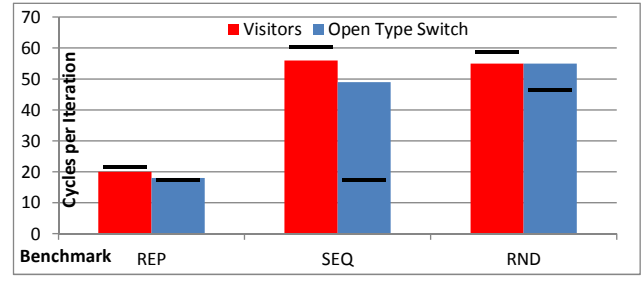To understand better the relative numbers of Figure 6, we present in Figure 5 few absolute timings taken by visi-

Figure 5. Absolute timings for different benchmarks

tors and open type switch to execute an iteration of a given benchmark. These absolute timings correspond to the relative numbers from column Open/G++/Win of Figure 6. The actual bars show the timings without forwarding, while the black lines indicate where the corresponding bar would be in the presence of forwarding. It is easy to see that visitors generally become slower in the presence of forwarding due to extra call, while type switch becomes faster due to smaller jump table. As discussed, both timings are much smaller for repetitive benchmark due to hardware cache.

Figure 6 provides a broader overview of how both techniques compare under different compiler/platform configurations. The values are given as percentages of performance increase against the slower technique.

		Open					Closed				
	G++		MS Visual C++				G++		MS Visual C++			
	Lnx	Win	PGO		w/o PGO		Lnx	Win	PGO		w/o PGO	
	x86-32	x86-32	x86-32	x86-64	x86-32	x86-64	x86-32	x86-32	x86-32	x86-64	x86-32	x86-64
REP	16%	14%	1%	**18%**	2%	**37%**	124%	122%	100%	41%	76%	37%
SEQ	56%	12%	48%	22%	**2%**	**46%**	640%	467%	29%	15%	30%	10%
RND	56%	0%	**9%**	**19%**	**5%**	**46%**	603%	470%	35%	20%	32%	6%
REP	33%	22%	8%	**17%**	24%	**36%**	53%	49%	24%	**11%**	20%	**36%**
SEQ	55%	233%	135%	135%	193%	**32%**	86%	290%	48%	139%	12%	**24%**
RND	78%	25%	3%	**4%**	13%	**23%**	88%	33%	8%	**1%**	18%	**16%**

Figure 6. Relative performance of type switching vs. visitors. Numbers in regular font (e.g. 14%), indicate when type switching was faster than visitors, while underlined numbers in bold (e.g. **18%**), indicate when visitors were faster by corresponding percentage.

We can see that type switching wins by a good margin when implemented with tag switch (§3.1) as well as in the presence of at least one level of forwarding. Note that the numbers are relative, and thus the ratio depends on both the performance of virtual function 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 x86-64 is only slower relatively: the actual time spent for both visitors and type switching was smaller than that for x86-32, but it was much smaller for visitors than type switching, which resulted in worse relative performance.

Lastly, 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 directives in GCC to guide the compiler, but, unfortunately, Visual C++ does not provide any similar facilities. Instead, Microsoft suggests using *Profile-Guided Optimizations* (PGO) to achieve the same, which is why we list the results for Visual C++ both with and without profile-guided optimizations.

4.2 Open vs. Closed Type Switch

With only a few exceptions, we saw in the Figure 6 that the performance of the closed tag switch dominates the performance of the open type switch. We believe that the difference, often significant, is the price one pays for the true openness of the vtable pointer memoization solution.

As we mentioned in §3.1, the use of tags, even when allocated by a 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 [22], 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 part of Figure 6 representing closed tag switch, since the tags in our experiments were all sequential and small.

The reliance of a tag switch on static cast has severe limitations in the presence of multiple inheritance, and thus is not as versatile as open type switch. Overcoming this problem will either require the use of **dynamic_cast** or techniques similar to vtable pointer memoization, which will likely degrade tag switch's performance numbers even further.

Note also that the approach used to implement open type switch can be used to implement both first-fit and best-fit semantics, while the tag switch is only suitable for best-fit semantics. Their complexity guarantees also differ: open type switch is constant on average, but slow on the first call with given subobject. Tag switch is logarithmic in the size of the class hierarchy (assuming a balanced hierarchy), including the first call. This last point can very well be seen in Figure 6, where the performance of a closed solution degrades significantly in the presence of forwarding, while the performance of an open solution improves.

4.3 Comparison with OCaml and Haskell

In this test, we timed small OCaml and Haskell applications 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 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 without a ReMatch loop. It is only supported through specialized syntax in our library as it differs from the Tag encoding only semantically.

We used the optimizing OCaml compiler `ocamlopt.opt` version 3.11.0 working under the Visual C++ toolset as well as the Glasgow Haskell Compiler version 7.0.3 (with -O

switch) working under the MinGW toolset. The C++ applications were compiled with Visual C++ as well and all the tests were performed on the Windows 7 laptop. Similar to comparison with visitors, the timing results presented in Figure 7 are averaged over 101 measurements and show the number of seconds it took to perform a 1,000,000 decompositions within our sequential benchmark. We compare here time and not cycles, as that was the only common measurement in all three environments.
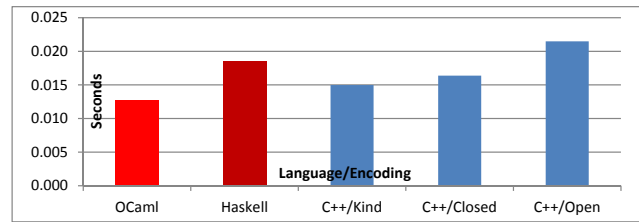

Figure 7. Performance comparison with OCaml & Haskell

4.4 Dealing with real-world class hierarchies

For this experiment, we used a class hierarchy benchmark previously used in the literature to study efficiency of type inclusion testing and dispatching techniques [17, 31, 50, 57]. We use the names of each benchmark from Vitek et al [50, Table 2], since the set of benchmarks we were working with was closest (though not exact) to that work.

While not all class hierarchies originated from C++, for this experiment it was more important for us that the hierarchies were man-made. While converting the hierarchies into C++, we had to prune inaccessible base classes (direct base class that is already an indirect base class) when used with repeated inheritance in order to satisfy semantic requirements of C++. We maintained the same number of virtual functions present in each class as well as the number of data members; the benchmarks, however, did not preserve the exact types of those. The data in Figure 8 shows various parameters of the class hierarchies in each benchmark, after their adoption to C++.

LIBRARY	LANGUAGE	CLASSES	PATHS	HEIGHT	ROOTS	LEAFS	BOTH	PARENTS		CHILDREN	
								AVG	MAX	AVG	MAX
DG2	SMALLTALK	534	534	11	2	381	1	1	1	3.48	59
DG3	SMALLTALK	1356	1356	13	2	923	1	1	1	3.13	142
ET+	C++	370	370	8	87	289	79	1	1	3.49	51
GEO	EIFFEL	1318	13798	14	1	732	0	1.89	16	4.75	323
JAV	JAVA	604	792	10	1	445	0	1.08	3	4.64	210
LOV	EIFFEL	436	1846	10	1	218	0	1.72	10	3.55	78
NXT	OBJECTIVE-C	310	310	7	2	246	1	1	1	4.81	142
SLF	SELF	1801	36420	17	51	1134	0	1.05	9	2.76	232
UNI	C++	613	633	9	147	481	117	1.02	2	3.61	39
VA2$_a$	SMALLTALK	3241	3241	14	1	2582	0	1	1	4.92	249
VA2$_k$	SMALLTALK	2320	2320	13	1	1868	0	1	1	5.13	240
VW1	SMALLTALK	387	387	9	1	246	0	1	1	2.74	87
VW2	SMALLTALK	1956	1956	15	1	1332	0	1	1	3.13	181
	OVERALLS	15246	63963	17	298	10877	199	1.11	16	3.89	323

Figure 8. Benchmark class hierarchies

The number of paths represents the number of distinct inheritance paths from the classes in the hierarchy to the roots of the hierarchy. This number reflects the number of possible subobjects in the hierarchy. The roots listed in the table are classes with no base classes. We will subsequently use the term *non-leaf* to refer to the possible root of a subhierarchy. Leafs are classes with no children, while *both* refers to utility classes that are both roots and leafs and thus neither have base nor derived classes. The average for the number of parents and the number of children were computed only among the classes having at least one parent or at least one child correspondingly.

With few useful exceptions, it generally makes sense to apply type switch only to non-leaf nodes of the class hierarchy. 71% of the classes in the entire benchmarks suite were leaf classes. Out of the 4369 non-leaf classes, 36% were spawning a subhierarchy of only 2 classes (including the root), 15% – a subhierarchy of 3 classes, 10% of 4, 7% of 5 and so forth. Turning this into a cumulative distribution, $a\%$ of subhierarchies had more than b classes in them:

a	1%	3%	5%	10%	20%	25%	50%	64%	100%
b	700	110	50	20	10	7	3	2	1

These numbers reflect the percentage of use cases one may expect in the real word that have a given number of case clauses in them.

For each non-leaf class A we created a function performing a type switch on every possible derived class D_i of it as well as itself. The function was then executed with every possible subobject $D_i \prec \sigma_j \succ A$ it can possibly be applied to, given the static type A of the subject. It was executed multiple but the same number of times on each subobject to ensure uniformity on one side (since we do not have the data about the actual probabilities of each subobject in the benchmark hierarchies) as well as let the type switch infer the optimal parameters k and l of its cache indexing function H_{kl}^V. We then plotted a point in chart of Figure 9 relating 2 characteristics of each of the 4396 type switches tested: the optimal computed probability of conflict p achieved by the type switch and the number of subobjects n that came through that type switch. The actual frequencies of collisions were within one tenth of a percentage point of the computed probabilities, which is why we did not use them in the chart. To account for the fact that multiple experiments could have resulted in the same pair (n, p), we use a shadow of each point to reflect somewhat the number of experiments yielding it.

The curves on which the results of experiments line up correspond to the fact that under uniform distribution of n subobjects, only a finite number of different values representing the probability of conflict p is possible. In particular, all such values $p = \frac{m}{n}$, where $0 \le m < n$. The number m reflects the number of subobjects an optimal cache indexing function H_{kl}^V could not allocate their own entry for and we showed in §3.8 that the probability of conflict under uniform distribution of n subobjects depends only on m. The curves thus correspond to graphs of functions $y = \frac{m}{x}$ for different
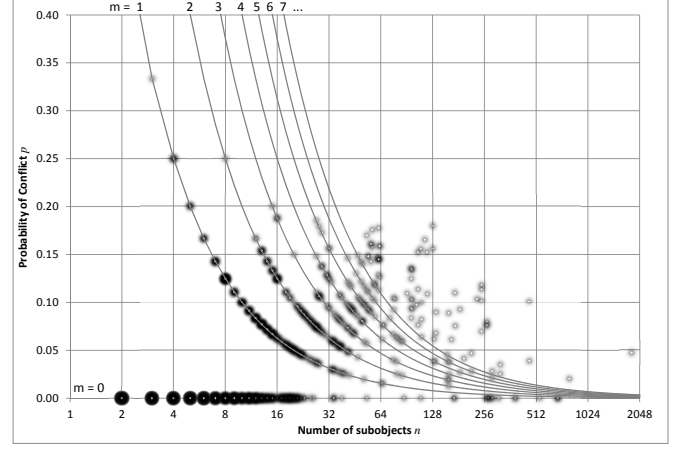

Figure 9. Probability of conflict in real hierarchies

values of m. The points on the same curve (which becomes a line on a log-log plot) all share the same number m of "extra" vtbl-pointers that optimal cache indexing function could not allocate individual entries for.

While it is hard to see from the chart, 87.5% of all the points on the chart lay on the X-axis, which means that the optimal hash function for the corresponding type switches had no conflicts at all ($m = 0$). In other words, only in 12.5% of cases the optimal H_{kl}^V for the set of vtbl-pointers V coming through a given type switch had non-zero probability of conflict. Experiments laying on the first curve amount to 5.58% of subhierarchies and represent the cases in which optimal H_{kl}^V had only one "extra" vtbl-pointer ($m = 1$). 2.63% of experiments had H_{kl}^V with 2 conflicts, 0.87% with 3 and so forth as shown in Figure 10($K + 1$).

m	0	1	2	3	4	5	6	>6
$K + 1$	87.50%	5.58%	2.63%	0.87%	0.69%	0.69%	0.30%	1.76%
K	72.55%	12.27%	4.87%	2.61%	1.42%	0.94%	0.80%	4.55%

Figure 10. Percentage of type switches with given number of conflicts (m) under different size constraints

In cases when the user is willing to trade performance for better space efficiency she may restrict k to $[K, K]$ instead of $[K, K + 1]$ as discussed in §3.8. We redid all the 4396 experiments under this restriction and obtained a similar histogram shown in Figure 10(K). The average probability of conflict over the entire set increased from 0.011 to 0.049, while the maximum probability of conflict increased from 0.333 to 0.375. The average load factor of the cache expectedly increased from 75.45% to 82.47%.

It is important to understand that the high ratio of cases in which the hash function could deliver perfect indexing does not indicate that the hash function we used is better than other hash functions. It does indicate instead that the values representing vtbl-pointers in a given application are not random at all and are particularly suitable for such a hash function.

4.5 Refactoring an existing visitors based application

For this experiment, we reimplemented a visitor based C++ pretty printer for Pivot[15] using *Mach7*. The Pivot's class hierarchy consists of 154 node kinds representing various entities in the C++ program. The original code had 8 visitor classes each handling 5, 7, 8, 10, 15, 17, 30 and 63 cases, which we turned into 8 match statements with corresponding numbers of case clauses. 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 type switching we needed them to come after due to first-fit semantics. Redundancy checking support provided by *Mach7* was invaluable in finding all such cases.

Both pretty printers were executed on a set of header files from the C++ standard library and the produced output of both program was byte-to-byte the same. We timed execution of the pretty printing phase (not including loading and termination of the application or parsing of the source file) and observed that on small files (e.g. those from C run-time library and few small C++ files) visitors-based implementation was faster because the total number of nodes in AST and thus calls did not justify our set-up calls. In particular, visitor-based implementation of the pretty printer was faster on files of 44–588 lines of code, with average 136 lines per those inputs, where visitors win. On these input files, visitors are faster by 1.17%–21.42% with an average speed-up of 8.75%. Open type switch based implementation of the pretty printer was faster on files of 144–9851 lines of code, with average 3497 lines per those input files, where open type switch wins. On these inputs, open type switch is faster by 0.18% – 32.99% with an average speed-up of 5.53%.

Figure 11 shows memory usage as well as cache hits and misses for the run of our pretty printer on ⟨queue⟩ standard library header, which had the largest LOC count after preprocessing in our test set.
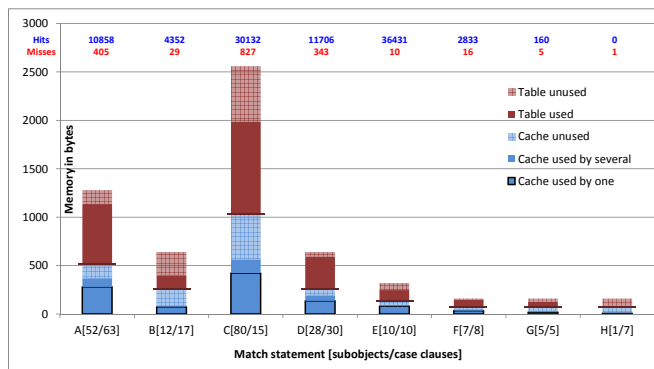

Figure 11. Memory usage in real application

The bars represent the total size of memory in bytes each of the 8 match statements (marked A-H) used. Information [n/c] next to the letter indicates the actual number of differ-ent subobjects (i.e. vtbl-pointers) n that came through that match statement, and the number of case clauses c the match statement had (the library uses c as an estimate of n). n is also the number of cases the corresponding match statement had to be executed sequentially (instead of a direct jump).

The lower part of each bar (with respect to dividing line) corresponds to the memory used by cache, while the upper part – to the memory used by the hash table. The ratio of the darker section of each part to the entire part indicates the load factors of cache and hash-table respectively. The black box additionally indicates the proportion of cache entries that are allocated for only one vtbl-pointer and thus never result in a cache miss.

The actual number of hits and misses for each of the match statements is indicated on top of the corresponding column. The sum of them is the total number of calls made. The number of misses is always larger than or equal to n since we need to execute the switch sequentially on each of them once in order to memoize the outcome.

The library always preallocates memory for at least 8 sub-objects to avoid unnecessary recomputations of optimal parameters k and l – this is the case with the last 3 match statements. In all other cases it allocates the memory proportional to 2^{K+1} where $2^{K-1} < \max(n,c) \le 2^K$. We make c a parameter, because in a library setting n is not known up front and estimating it with c allows us to avoid unnecessary recomputations of l and k even further.

The table does not have to be hash table and can be implemented with any other container i.e. sorted vector, map etc. that let us find quickly by a given vtbl-pointer the data associated with it. In fact we provide a slightly less efficient caching container that avoids the table altogether, thus significantly reducing the memory requirements instead.

4.6 Limitations

Currently the definition of each class used in a case clause must be visible to the compiler because **dynamic_cast** operator used in the type switch does not allow incomplete types as a target type. For particularly large type switches (e.g. >1000 case clauses) this may easily reach some compiler limitations. Both GCC and Visual C++, for example, could not generate object files for such translation units simply because the sheer size of v-tables and other compiler data in it were exceeding the limits. The problem is not specific to our technique though and allowing **dynamic_cast** on classes that were declared but not defined yet would solve the problem.

While it might be reasonable to expect from linkers to layout v-tables close to each other – the property that makes our hashing function efficient – they are not required to do so. We believe, nevertheless, that should our approach become popular through the library implementation, its compiler implementation will encourage compiler vendors to enforce the property in order to keep the type switching fast.

5.   Related Work

Lookup caches have been long used to reduce the overhead of dynamically-bound message passing in Smalltalk [49]. *Inline caching* improves on that by noting that the type of the receiver at a given call site rarely varies so that the call instruction can be speculatively modified to jump directly to a previously looked up method [13]. In this case, the method must ensure that the type of the receiver has not changed and redirect the call to generic lookup otherwise. The effects of inline caching on modern architectures can be seen through hardware call target prediction, which in our case is exemplified by repetitive benchmark: both virtual function calls and the underlying jump-table implementation of the *Match*-statement are about twice as fast as usual.

Polymorphic Inline Caches [27] generalize the idea of inline caches further by building a decision tree in the method prologue that caches all lookup results. The main difference of this approach from our work is that it requires code generation at run time, while we do not require re-compilation, relinking or any computations in case of dynamic linking. The reason for this is the difference in the initial setting: they map an arbitrary number of receiver types to an arbitrary number of implementations, while we map an arbitrary number of receivers to a fixed number of jump targets. This lets us generate code at compile time that incorporates both the initial and memoized execution.

Extensible Visitors with Defaults [55, §4.2] offer a solution to the extensibility problem of visitors. The visitation interface hierarchy can easily be grown linearly, but independent extensions by different programmers require manual coordination. In addition to the double dispatch, the solution incurs two additional virtual calls and a dynamic cast for each level of visitor extension. The solution is simpler with virtual inheritance, which adds even more indirections.

Löh and Hinze proposed to extend Haskell's type system with open data types and open functions [35]. The solution allows top-level data types and functions to be marked as open with concrete variants and overloads defined anywhere in the program. The semantics of open extension is given by transformation into a single module, which assumes a whole-program view and thus is not an open solution by our definition. Besides, open data types are extensible but not hierarchical, which avoids the problems discussed here.

Andrew Kennedy et al [30] considered encoding of generalized algebraic data types [42] using visitor design patterns in C#. That translation made essential use of generic methods, the equivalent of "virtual function template" (as they would be called if such functionalities existed in C++) to handle some of the open set aspects of GADTs.

Polymorphic variants in OCaml [22] allow the addition of new variants, while defining subtyping on them. The subtyping, however, is not defined between the variants, but between combinations of them. This maintains disjointness between values from different variants and makes an important distinction between *extensible sum types* like polymorphic variants and *extensible hierarchical sum types* like classes. Our memoization device can be used to implement pattern matching on polymorphic variants as well.

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 [38]. Its type patterns and %match statement can be used as a type switch, however, their implementation is based on decision trees and an instanceof-like predicate.

Pattern matching in Scala [39] also supports type switching through type patterns. 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, while extensible classes are handled with a combination of an InstanceOf operator and a decision tree [19].

6.   Conclusions and Future Work

Type switching is an open alternative to the visitor design pattern that overcomes the restrictions, inconveniences, and difficulties in teaching and using visitors. Our implementation significantly outperforms the visitor design pattern in most cases and roughly equals it otherwise. This is the case even though we use a library implementation and highly optimized production-quality compilers. An important benefit of our solution is that it does not require any changes to the C++ object-model or require any computations at load time.

To provide a complete solution, we use the same syntax for closed sets of types, where our performance roughly equals the equivalent built-in features in functional languages, such as Haskell and OCaml.

We prove the uniqueness of vtbl-pointers in the presence of RTTI. This is potentially useful in other compiler optimizations that depend on the identity of subobjects. Our memoization device can also become valuable in optimizations that require mapping run-time values to execution paths, and is especially useful in library setting.

Using a library implementation was essential for experimentation and for being able to test our ideas on multiple production-quality compiler systems. However, now we hope to re-implement our ideas in a compiler. This would allow us to improve further surface syntax, diagnostics, and performance.

References

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

[2] Liz: A System for Axiomatic Programming. http://liz.axiomatics.org/, 2012.

[3] 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.

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

[5] L. Cardelli. Compiling a functional language. In *Proc. of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 208–217, New York, NY, USA, 1984. ACM.

[6] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The Modula-3 type system. In *Proc. of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 202–212, New York, NY, USA, 1989. ACM.

[7] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In *Proc. of the 8th conference on Object-oriented programming systems, languages, and applications*, OOPSLA '93, pages 271–287, New York, NY, USA, 1993. ACM.

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

[9] N. H. Cohen. Type-extension type test can be performed in constant time. *ACM Trans. Program. Lang. Syst.*, 13(4):626–629, Oct. 1991.

[10] W. R. Cook. Object-oriented programming versus abstract data types. In *Proc. of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 151–178, London, UK, 1991. Springer-Verlag.

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

[12] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: an optimizing compiler for object-oriented languages. In *Proc. of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, pages 83–100, New York, NY, USA, 1996.

[13] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proc. of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[14] E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. Jan. 1975.

[15] G. Dos 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.

[16] K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. In *Proc. of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, pages 306–323, New York, NY, USA, 1996. ACM.

[17] R. Ducournau. Perfect hashing as an almost perfect subtype test. *ACM Trans. Program. Lang. Syst.*, 30(6):33:1–33:56, Oct. 2008. ISSN 0164-0925.

[18] M. A. Ellis and B. Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. ISBN 0-201-51459-1.

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

[20] 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.

[21] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proc. of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 406–431, London, UK, UK, 1993. Springer-Verlag.

[22] J. Garrigue. Programming with polymorphic variants. In *ACM Workshop on ML*, Sept. 1998.

[23] M. Gibbs and B. Stroustrup. Fast dynamic casting. *Softw. Pract. Exper.*, 36:139–156, February 2006.

[24] N. Glew. Type dispatch for named hierarchical types. In *Proc. of the 4th ACM SIGPLAN international conference on Functional programming*, ICFP '99, pages 172–182, New York, NY, USA, 1999.

[25] C. Grothoff. Walkabout revisited: The runabout. In *ECOOP 2003 - Object-Oriented Programming*, pages 103–125. Springer-Verlag, 2003.

[26] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 130–141, New York, NY, USA, 1995. ACM.

[27] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proc. of the European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 21–38, Berlin, Germany, 1991. Springer-Verlag.

[28] International Organization for Standardization. *ISO/IEC 14882:2011: Programming languages: C++*. Geneva, Switzerland, 3rd edition, 2011.

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

[30] A. Kennedy and C. V. Russo. Generalized algebraic data types and object-oriented programming. In *Proc. of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 21–40, New York, NY, USA, 2005. ACM.

[31] A. Krall, J. Vitek, and R. N. Horspool. Near optimal hierarchical encoding of types. In *In Proc. European Conference on Object-Oriented Programming, ECOOP'97, Lecture Notes in Computer Science*, pages 128–145. Springer-Verlag, 1997.

[32] 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.

[33] B. Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proc. on Object-oriented programming systems, languages and applications*, pages 17–34, New York, NY, USA, 1987. ACM Press.

[34] B. Liskov, R. R. Atkinson, T. Bloom, E. B. Moss, R. Schaffert, and A. Snyder. Clu reference manual. Technical report, Cambridge, MA, USA, 1979.

[35] A. Löh and R. Hinze. Open data types and open functions. In *Proc. 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.

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

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

[38] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *Proc. of the 12th international conference on Compiler construction*, CC'03, pages 61–76, Berlin, Heidelberg, 2003. Springer-Verlag.

[39] M. Odersky, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. Mcdirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. An overview of the Scala programming language (2nd edition). Technical Report LAMP-REPORT-2006-001, Ecole Polytechnique Federale de Lausanne, 2006.

[40] B. C. Oliveira, M. Wang, and J. Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *Proc. of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 439–456, New York, NY, USA, 2008. ACM.

[41] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Proc. of the 22nd International Computer Software and Applications Conference*, COMPSAC '98, pages 9–15, Washington, DC, USA, 1998. IEEE Computer Society.

[42] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proc. of the 11th ACM SIGPLAN international conference on Functional programming*, ICFP '06, pages 50–61, New York, NY, USA, 2006. ACM.

[43] T. Ramananandro, G. Dos Reis, and X. Leroy. Formal verification of object layout for C++ multiple inheritance. In *Proc. of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 67–80, New York, NY, USA, 2011. ACM.

[44] J. G. Rossie, Jr. and D. P. Friedman. An algebraic semantics of subobjects. In *Proc. of the 10th conference on Object-oriented programming systems, languages, and applications*, OOPSLA '95, pages 187–199, New York, NY, USA, 1995. ACM.

[45] L. Schubert, M. Papalaskaris, and J. Taugher. Determining type, part, color, and time relationships. *Computer*, 16:53–60, 1983. ISSN 0018-9162.

[46] 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.

[47] B. Stroustrup. Multiple inheritance for C++. In *Proc. of the Spring'87 EUUG Conference*, EUUG '87, May 1987. Revised version in AT&T C++ Translator Release Notes, June 1989. Also, USENIX Computing Systems, V2 no 4, Fall 1989, pp 367-396.

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

[49] D. Ungar and D. Patterson. Berkeley smalltalk: Who knows where the time goes? In G. Krasner, editor, *Smalltalk-80: Bits of History and Words of Advice*, pages 189–206. Addison-Wesley, 1983. ISBN 0-201-11669-3.

[50] J. Vitek, R. N. Horspool, and A. Krall. Efficient type inclusion tests. In *Proc. of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '97, pages 142–157, New York, NY, USA, 1997. ACM.

[51] D. Vytiniotis, G. Washburn, and S. Weirich. An open and shut typecase. In *Proc. of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '05, pages 13–24, New York, NY, USA, 2005. ACM.

[52] P. Wadler. The expression problem. Mail to the java-genericity mailing list, November 1998.

[53] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *Proc. 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.

[54] N. Wirth. Type extensions. *ACM Trans. Program. Lang. Syst.*, 10(2):204–214, Apr. 1988.

[55] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proc. of the 6th ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 241–252, New York, NY, USA, 2001. ACM.

[56] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *Proc. FOOL 12*, Jan. 2005.

[57] Y. Zibin and J. Y. Gil. Efficient subtyping tests with PQ-encoding. In *Proc. of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 96–107, New York, NY, USA, 2001. ACM.