Date: 3/19/2016
Reply to: Andrea Proli
(andy.prowl at gmail dot com)

Dynamic Generic Programming with Virtual Concepts

Author: Andrea Proli

Abstract

The present proposal aims at introducing language support for type erasure in C++ in the form of so-called *virtual concepts*. Virtual concepts allow extending the set of interfaces a type adheres to without modifying its definition, and without requiring generic algorithms and data types that rely on those interfaces to be written as templates.

The proposed solution works both for user-defined types and for fundamental types, and does not require user-defined types to be polymorphic – i.e. they may have no virtual member functions. Adherence of a type to a concept can be either deduced (through *duck-typing*) or asserted (through *concept maps*).

Both value semantics and reference semantics are supported. Supporting value semantics means allowing the manipulation of erased types through the typical operations on value types (assignment, copy- and move-construction, equality comparison, etc.); supporting reference semantics means allowing the manipulation of erased types through native references and pointers, without requiring any sort of proxy or wrapper.

In terms of efficiency, the proposed solution yields *in the worst case* the same overhead as inheritance-based polymorphism for the use cases that the latter supports – namely, reference semantics – both memory-wise and performance-wise.

Virtual concepts offer a simple alternative to type erasure techniques which are usually hard to implement, limited in expressivity, and/or make code hard to read and slow to build. In particular, virtual concepts simplify and generalize common library utilities such as `std::any` and `std::function`.

Finally, virtual concepts provide a native and elegant solution to the Expression Problem [1] and to the Proxy Dilemma [2] in C++.

Table of Contents

Table of Figures

1. Introduction

Generic Programming (GP) is a programming paradigm which encourages the formulation of algorithms in terms of a minimal set of abstract requirements on the types of the objects they operate on. This set of requirements, which is said to define a *concept*, takes the role of a contract regulating the interaction between the generic algorithm and its inputs.

At a *logical* level, concepts represent the layer of indirection which allows separating high-level, abstract operations from type-specific implementation details, breaking the dependency of the former on the latter by making both depend on an intermediate entity:

Figure 1 – Abstraction role of concepts in Generic Programming

This logical layer of indirection makes it possible to modify the behavior of an algorithm by substituting any of the concrete types it operates on for a different type *without requiring any change to the definition of the algorithm itself*, as long as the new type satisfies the corresponding concept. The generic algorithm becomes therefore a piece of highly reusable software.

In C++, GP is most often achieved through the use of templates. Templates offer a mechanism for parameterizing algorithms (i.e. function templates) and data structures (i.e. class templates) with type and non-type arguments that are known *at compile-time* – which is why this mechanism is commonly referred to as *static polymorphism*.

The main benefits of using templates for GP are:

1. *Composability*: The modeling relation between a type T and a concept C is *deduced* by the compiler based on syntactic conformance of T's interface to C's requirements; conformance is not asserted on T's definition. Consequently, making T a model of a *new* concept D will not require changing the definition of T;
2. *Value semantics*: Unlike inheritance, templates do not require generic algorithms to access their arguments through pointers or references. This is in line with C++'s philosophy of supporting user-defined types that "behave like an `int`". Reference semantics is, however, not precluded;
3. *Efficiency*: Since templates are instantiated at compile-time, and the definition of type arguments must be made available to the compiler when instantiating a generic algorithm, the compiler has enough information to inline function calls inside the algorithm's body; this results in machine code which is often as fast as (or even faster than) the one produced by a hand-written, non-generic algorithm.

The last point is a double-edged sword: requiring an argument type T's definition to be available at compile-time to the generic algorithm means that (1) changes to T's definition will require recompiling all the translation units which provide objects of type T as an input to a generic algorithm (including the algorithm's code) and (2) the algorithm's definition cannot be kept in a separate translation unit. At a *physical* level, the independence schematized in Figure 1 no longer holds:

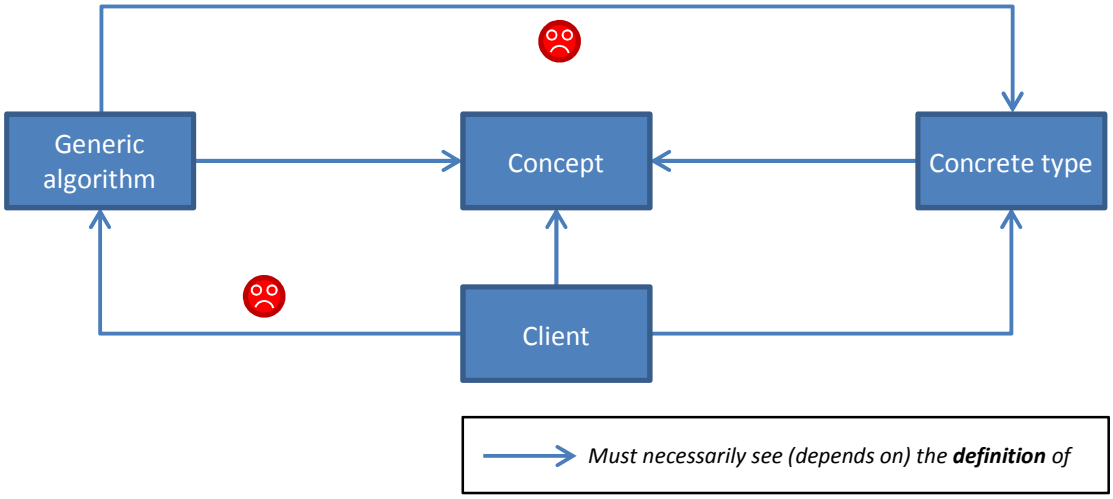

Figure 2 – Physical dependency schema for static polymorphism based on templates

If independent compilation and deployment is important, the programmer is forced to resort to alternative techniques. The most popular among such techniques in OOP languages is inheritance-based *dependency inversion* [3], which allows execution control to flow from the generic algorithm to the concrete type at run-time without the former being physically dependent on the latter:

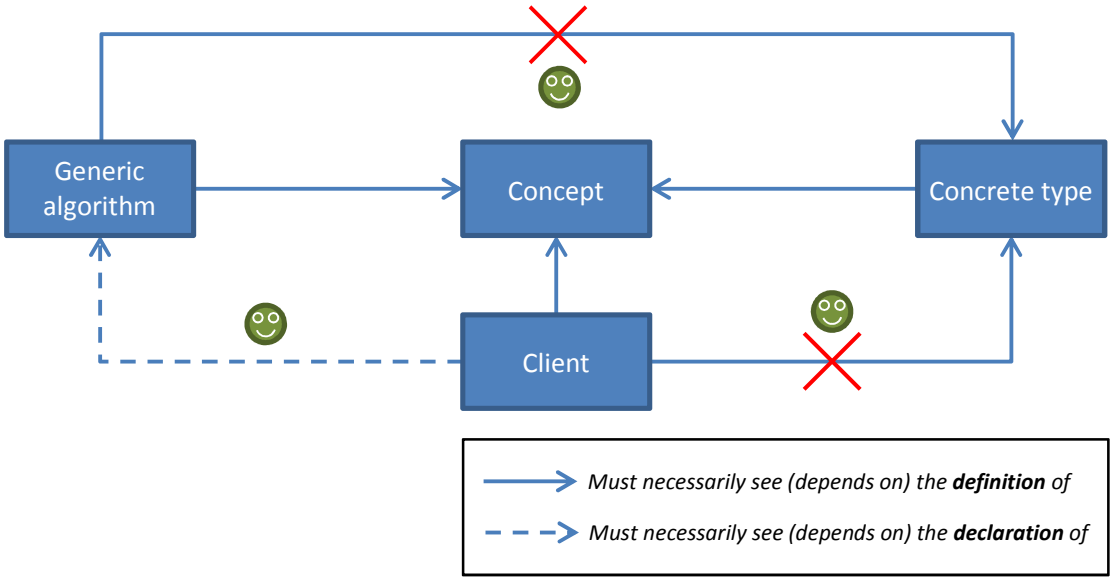

Figure 3 – Physical dependency schema for dynamic polymorphism based on inheritance

While supporting separate compilation, inheritance-based polymorphism has several shortcomings:

1. *Intrusiveness*: the adherence of a type to a contract must be mentioned in that type's definition. This is a problem for extensibility: if we wanted to make an existing type adhere to a *new* contract in order to enable polymorphic dispatch, we would have to change that type's definition – which may be troublesome or not possible;
2. *Applicability*: inheritance-based polymorphism is limited to user-defined types, because the language forbids making a fundamental type (e.g. `int`) derive from an abstract class to implement its interface. This means we cannot provide an object of a fundamental type to a generic algorithm defined in terms of an abstract interface;
3. *Reference semantics*: inheritance-based polymorphism forces on the programmer the use of pointers or references to access objects of polymorphic types. While value semantics can be achieved, the adoption of sophisticated (read: complicated) techniques is inevitable [4] [5];
4. *Performance*: Run-time function call binding prevents the compiler from performing precious optimizations such as function call inlining.

Of all these drawbacks, only the last one is inherent to dynamic polymorphism *as a pattern*: the previous ones are characteristic of inheritance *as a technique*.

This proposal aims at introducing language support for type erasure in C++ that would allow for a run-time form of GP – which I refer to as *Dynamic Generic Programming* (DGP) – in a way similar to how templates and concepts [6] support compile-time GP.

The proposed solution, based on what I call *virtual concepts*, provides the composability and flexibility of classical compile-time GP, while retaining the advantages of inheritance in terms of independent deployability. The price to be paid in exchange for these benefits is the inherent performance penalty of dynamic binding, but no more than that – see Section 3 for a detailed account on our design goals.

The rest of this document is structured as follows: Section 2 provides an overview of the current state-of-the-art approaches to DGP in C++; Section 3 states the design goals which underlie and shape our proposal; Section 4 gives a brief demonstration of the capabilities of virtual concepts, showing how they overcome the limitations of existing DGP techniques; Section 5 shows how to define a virtual concept, and how to extract a type-erasing interface definition from the associated requirements; Section 6 covers the details of how virtual concepts support value semantics and reference semantics; Section 7 illustrates the mechanisms of *duck-typing* and *concept maps* to express the modeling relationship between a type and a virtual concept; Section 8 introduces the fundamental relationships between concepts, such as *refinement* and *subsumption*, and discusses their decidability; Section 9 elaborates on the interplay between virtual concepts, templates, and inheritance; Section 10 focuses on the modifications required to C++'s overload resolution algorithm; Section 11 explains how to revert the type-erasing procedure through *concept casting*; Section 12 presents a possible implementation technique and reports on implementation experience; finally, Section 13 draw the conclusions.

2. Existing approaches to DGP

The C++ Standard Library provides types such as `std::function`, for type-erasing callable objects compatible with a given signature, and `std::any`, for type-erasing objects of any type; `std::shared_ptr` offers support for type-erasing custom deleters. However, neither the Standard Library nor the core language provides general-purpose tools for defining new type-erasing wrappers.

Several attempts have been made at filling this gap at a library level. An excellent overview of type-erasure techniques is given in [7] (parts I through IV), including a comparison between two of the most popular publicly-available library solutions: Adobe.Poly [8] and Boost.TypeErasure [5].

An interesting recent work is Pyry Jahkola's C++11 Poly library [9], which is nevertheless not meant for production use. Also worth mentioning is the Boost.Interfaces library [10] (which is *not* part of Boost [11] and seems to be no longer maintained) as one of the earliest attempts to support DGP in C++.

A radically different approach, addressing the problem at a language level, was chosen for the Signatures project [12], implemented as a language extension for GCC (up to version 2.95) [13] [14].

I shall now set up a working example that will help us evaluate and compare these solutions.

2.1. Working example

Let's define two classes, `Circle` and `Rectangle`, as shown below. Both classes have member functions `get_area()` and `get_perimeter()` that return a `double`. These classes do not derive from any common base class, and we will assume them to be part of a closed-source, third-party static library we are linking into our project. This implies that we are not allowed to modify their definitions:

```
struct Circle
{
    Circle();
    Circle(Point center, double radius);
    double get_area() const;
    double get_perimeter() const;
private:
    // ...
};

struct Rectangle
{
    Rectangle();
    Rectangle(Point center, double length, double width);
    double get_area() const;
    double get_perimeter() const;
private:
    // ...
};
```

Our goal is to support the definition of the generic algorithm `print_areas()` shown below:

```
void print_areas(std::vector<Shape> const& v)
{
    for (auto const& s : v)
    {
        std::cout << s.get_area() << " ";
    }
}
```

`print_areas()` accepts in input a heterogenous vector of `Shape` objects, where `Shape` is some erasing wrapper that is capable of representing any object whose type satisfies the associated concept's requirements.

For our purposes, the concept of a *shape* requires a conforming type to offer non-mutating functions `get_area()` and `get_perimeter()` that return, respectively, the area and the perimeter of the object. It is easy to verify that both classes `Circle` and `Rectangle` satisfy these requirements.

Here is how the requirements could be expressed using the syntax proposed by the Concepts TS [6]:

```
template<typename T>
concept bool Shape = requires(T const& x)
{
    { x.get_area() } -> double;
    { x.get_perimeter() } -> double;
    requires SemiRegular<T>;
};
```

The obvious constraint that makes this problem hard is that the translation unit in which `print_areas()` is defined may not include the header files containing the definitions of `Circle` and `Rectangle`; yet, `print_areas()` must be able to dispatch function calls to instances of those types through the `Shape` wrapper.

Furthermore, we expect the program below to be well-formed and to yield the correct output (assuming all the necessary object files are linked in, and omitting standard header inclusions):

```
#include "shape.hpp" // CONTAINS THE DEFINITION OF THE TYPE-ERASING WRAPPER

void print_areas(std::vector<Shape> const& v);

int main()
{
    auto r = Rectangle{{1.0, 2.0}, 5.0, 6.0};

    auto c = Circle{{0.0, 0.0}, 42.0};

    std::vector<Shape> v{Shape{r}, Shape{c}};

    print_areas(v);
}
```

2.2. Adobe.Poly

Adobe.Poly [8] is part of the open-source Adobe Source Libraries (ASL) [15]. The goal of Adobe.Poly is to allow "*creating regular, runtime-polymorphic objects with value semantics*" [15]. The philosophy and theoretical underpinnings of Adobe.Poly are discussed in detail in [4], [16], [17], and [18]. The source code for ASL can be found at http://sourceforge.net/projects/adobe-source/.

Adobe.Poly, which predates the Concepts TS, does not allow generating type-erasing wrappers directly from concept definitions like the one given in Section 2.1. Instead, it requires the user to define (for each concept) one abstract interface, one class template, and one concrete class; utilities provided by the framework help reducing the amount of boilerplate code that the user is required to type.

We will start with the abstract interface, which is *not* meant to be used directly by generic algorithms like `print_areas()`; instead, it is used internally for dispatching function calls to the erased types. It defines the common operations that the types satisfying our concept must support, but does not require their signatures to match exactly those in the erased types, nor to match those of the Shape concept; not even a one-to-one correspondence between these functions is necessary, albeit common.

Here is how we could define our `ShapeInternal` interface:

```
struct ShapeInternal : adobe::poly_copyable_interface
{
    virtual double get_area_internal() const = 0;
    virtual double get_perimeter_internal() const = 0;
};
```

The abstract base class `poly_copyable_interface` is part of ASL, and defines common operations on copyable types as well as other functions used by the framework.

Next, we need to define a generic holder for our type-erased objects. The holder is parameterized by the type of the wrapped object, and implements `ShapeInternal` by delegating to it:

```
template<typename T>
struct ShapeHolder : adobe::optimized_storage_type<T, ShapeInternal>::type
{
    using base =
          typename adobe::optimized_storage_type<T, ShapeInternal>::type;

    ShapeHolder(adobe::move_from<ShapeHolder> x)
        : base{adobe::move_from<base>{x.source}} { }

    ShapeHolder(T x) : base_t{std::move(x)} { }

    virtual double get_area_internal() const override
    { return this->get().get_area(); }

    virtual double get_perimeter_internal() const override
    { return this->get().get_perimeter(); }
};
```

The relevant portions of the definition of `ShapeHolder` are the constructor accepting an object of type `T`, which eventually gets stored by the holder's base constructor, and the delegating implementation of `get_area_internal()` and `get_perimeter_internal()` – everything else is for integration with the framework. Also notice, that the base class template `optimized_storage_type` – which implements the Small Buffer Optimization (SBO) – must be instantiated with the type of the abstract interface (`ShapeInternal`).

Finally, the concrete class: this is the type which defines the interface of the concept, i.e. the operations that generic algorithms can invoke on the erased objects. We will call ours `ShapeEraser`. The actual erasing wrapper used by generic algorithms like `print_areas()` can be obtained by instantiating the `adobe::poly` class template with the eraser type:

```
struct ShapeEraser : adobe::poly_base<ShapeInternal, ShapeHolder>
{
    using base = adobe::poly_base<ShapeInternal, ShapeHolder>;

    using base::base; // INHERIT BASE CLASS'S CONSTRUCTORS

    ShapeEraser(adobe::move_from<ShapeEraser> x)
        : base_t{adobe::move_from<base_t>(x.source)} { }

    double get_area() const
    { return interface_ref().get_area_internal(); }

    double get_perimeter() const
    { return interface_ref().get_perimeter_internal(); }
};

using Shape = adobe::poly<ErasedShape>;
```

With all of this in place, the test code from Section 2.1 can compile and run correctly.

2.2.1. Adaptation vs. duck-typing

In the working example from Section 2.1, classes `Circle` and `Rectangle` have a syntactically uniform interface: in other words, the operations which return the area and perimeter of an object are realized by member functions with identical signature.

This made our life simpler when implementing the dispatch mechanism, because it allowed us to write a single class template (`ShapeHolder`) for delegating the implementation of those logical operations to any of our concrete types.

Had we attempted to use a `Shape` wrapper to type-erase an object with differently-named member functions, the result would have been a (likely cryptic) compiler error.

Consider, for instance, the following `Triangle` class, whose functions `calculate_area()` and `calculate_perimeter()` do not keep up with our current `ShapeHolder`'s expectations:

```
struct Triangle
{
    Triangle(Point v1, Point v2, Point v3);
    double calculate_area() const; // NON-CONFORMING NAME!
    double calculate_perimeter() const; // NON-CONFORMING NAME!
private:
    Point _v1;
    Point _v2;
    Point _v3;
};
```

Without further intervention, the code below would fail to compile:

```
auto t = Triangle{{0.0, 0.0}, {1.0, 0.0}, {0.0, 1.0}};

Shape s = t; // ERROR!
```

The approach to determining whether a type models a concept based exclusively on syntactic conformance is called *duck-typing*. Duck-typing has advantages and disadvantages depending on the use case, but Adobe.Poly allows performing type-specific adaptation by specializing the `ShapeHolder` class template for concrete types:

```
template<>
struct ShapeHolder<Circle>
    : adobe::optimized_storage_type<Circle, ShapeInternal>::type
{
    using base_t =
          typename adobe::optimized_storage_type<Circle, ShapeInternal>::type;

    ShapeHolder(Circle x) : base_t{std::move(x)} { }

    ShapeHolder(adobe::move_from<ShapeHolder> x)
        : base_t{adobe::move_from<base_t>{x.source}} { }

    virtual double get_area_internal() const override
    { return this->get().calculate_area(); }

    virtual double get_perimeter_internal() const override
    { return this->get().calculate_perimeter(); }
};
```

Adaptation provides a lot of flexibility when expressing the modeling relation between a type and a concept: in particular, types can be freely designed without worrying about the (possibly conflicting) syntactic requirements of all the generic algorithms that may use them.

2.2.2. Reference semantics

Adobe.Poly was designed to support DGP for types used with value semantics, and does not offer any support for reference semantics off-the-shelf. However, reference semantics use cases can be covered with some additional adaptation work.

In particular, to support reference semantics for a given concept, it is possible to define a new holder class template for instances of `std::reference_wrapper`, with proper adaptation performed in the function calls that delegate to the wrapped object. In the case of `ShapeHolderRef` below, the delegating functions `get_area_internal()` and `get_perimeter_internal()` would have to be rewritten so that `reference_wrapper`'s `get()` member function is called to access the object:

```
template<typename T>
struct ShapeHolderRef; // INTENTIONALLY LEFT UNDEFINED

template<typename T>
struct ShapeHolderRef<std::reference_wrapper<T>>
    : adobe::optimized_storage_type<std::reference_wrapper<T>,
                                    ShapeInternal>::type
{
    using base_t = typename adobe::optimized_storage_type<
        std::reference_wrapper<T>,
        ShapeInternal>::type;

    ShapeHolderRef(std::reference_wrapper<T> x) : base_t{std::move(x)} { }

    ShapeHolderRef(adobe::move_from<ShapeHolderRef> x)
        : base_t{adobe::move_from<base_t>{x.source}} { }

    virtual double get_area_internal() const override
    { return this->get().get().get_area(); }

    virtual double get_perimeter_internal() const override
    { return this->get().get().get_perimeter(); }
};
```

We could now write a new `ErasedShapeRef` wrapper; however, to reduce duplication, it makes sense to turn the existing `ShapeEraser` class into a class *template*, parameterized by the holder type:

```
template<template<typename> class HOLDER>
struct ShapeEraser : adobe::poly_base<ShapeInternal, HOLDER>
{
    using base_t = adobe::poly_base<ShapeInternal, HOLDER>;

    using base_t::base_t; // INHERIT BASE CLASS'S CONSTRUCTORS

    ShapeEraser(adobe::move_from<ShapeEraser> x)
    : base_t(adobe::move_from<base_t>(x.source)) { }

    // SAME AS BEFORE...
};
```

This allows defining the final type-erasing wrappers as follows:

```
using Shape = adobe::poly<ShapeEraser<ShapeHolder>>; // FOR VALUE SEMANTICS

using ShapeRef = adobe::poly<ShapeEraser<ShapeHolderRef>>; // FOR REF. SEM.
```

This allows writing code like the following:

```
int main()
{
    Rectangle r{{1.0, 2.0}, 5.0, 6.0};

    ShapeRef s{std::ref(r)};

    std::cout << s.get_area() << std::endl; // PRINTS 30

    r.set_size(4, 2); // SUPPOSE SUCH A FUNCTION EXISTS

    std::cout << s.get_area() << std::endl; // PRINTS 8
}
```

Due to how the `adobe::poly` base class works, the call to `std::ref()` above cannot be omitted – although simple modifications to the library's source code would allow doing that.

A worse problem is that if a certain type U needs dedicated adaptation logic – like the `Triangle` class from Section 2.2.1 – one additional explicit specialization of `ShapeHolderRef` for `std::reference_wrapper<U>` needs to be provided, which complicates maintenance.

Another issue is the lack of support for rvalue references: `std::reference_wrapper` is meant to simulate lvalue references only, and its constructor accepting an rvalue reference is deleted. Hence, creating a new eraser `ShapeRefRef` would be problematic and may require changes to the library's source code.

Finally, Adobe.Poly suffers from a problem which is common to all library solutions for DGP, at least concerning reference semantics: the Proxy Dilemma [2]. The problem stems from the fact that a referencing type-erasure wrapper is itself a *distinct object* from the object it erases. In other words:

```
auto r = Rectangle{{1.0, 2.0}, 5.0, 6.0};

auto s = ShapeRef{std::ref(r)};

assert(&r == &s); // THIS ASSERTION ALWAYS FAILS
```

This is an issue for compile-time generic algorithms written in the form of function templates: depending on how they are written, these algorithms may not be allowed to work transparently with objects accessed through a type-erasing wrapper. This situation is also described in [19] as part of the documentation of the Boost.TypeErasure library [5].

A related problem becomes apparent when trying to use smart pointers to type-erasing wrappers. For symmetry, one would expect the following initializations to be valid:

```
Shape s = Rectangle{{1.0, 2.0}, 5.0, 6.0};

std::unique_ptr<Shape> s = std::make_unique<Rectangle>({1.0, 2.0}, 5.0, 6.0);
```

Unfortunately, the second initialization is illegal, because the `unique_ptr` would internally hold a pointer to a `Shape`, which is not compatible with a pointer to a `Rectangle` (at a language level, those two types are unrelated). This can be worked around by creating a `Shape` object instead of a `Rectangle` object, but doing so will not help when we already have a `unique_ptr<Rectangle>` to begin with, and we want to pass it to a generic algorithm expecting a `unique_ptr<Shape>`.

2.2.3. Conclusions on Adobe.Poly

Adobe.Poly enables DGP for types with value semantics. While elegant and efficient – thanks to techniques such as SBO – it is not very easy to master, and requires writing a significant amount of boilerplate code, as shown in the previous sections.

This boilerplate code may be non-trivial due to the presence of templates and (possibly) template specializations; furthermore, it tends to become more complex when erasing overloaded operators such as the equality-comparison operator. For example, this is how our `ShapeHolder` would dispatch equality checks to the underlying object:

```
virtual bool equals(ShapeInternal const& rhs) const override
{ return this->type_info() == rhs.type_info() &&
         this->get() == *static_cast<const T*>(rhs.cast()); }
```

Support for reference semantics with wrappers such as `std::reference_wrapper`, while not provided off-the-shelf, can be by obtained at the cost of writing more boilerplate code; the amount of work starts to become a serious maintenance burden when adaptation is used instead of duck-typing.

Supporting reference semantics for rvalue references may not be possible without changing the library's code, and more inherent problems like the Proxy Dilemma [2] cannot be solved at all, limiting the possibility of mixing DGP with classical, compile-time GP.

Concerning build times, Adobe.Poly is a relatively light-weight library, because its internals are not heavily based on template meta-programming (TMP) techniques; yet, a certain overhead with respect to inheritance-based polymorphism is unavoidable.

As a final remark, although the documentation on the project's website does cover the details of the API to some degree, no introductory example or tutorial is available. This makes it very hard for a neophyte to understand how the library works, and how to make proper use of it.

2.3. Boost.TypeErasure

Boost.TypeErasure [5], authored by Steven Watanabe, is a recent addition to the Boost [11] suite of libraries, which first introduced it in version 1.54.

Thanks to a heavy use of TMP techniques and C-style macros, Boost.TypeErasure helps reducing the amount of boilerplate code users have to write in order to define their own type-erasing wrappers. However, this also means that compilation times are slower than those obtained with Adobe.Poly, and significantly slower than those obtained by using classical, inheritance-based polymorphism.

Another major disadvantage brought by TMP techniques is the quality of compiler diagnostics. Forgetting to specify a `const` qualifier where the framework expects it may result in pages of unintelligible error messages.

It is also worth mentioning that the amount of work needed to define custom concepts increases significantly when the advanced features of the library are used. One such situation is when adaptation is desired rather than duck-typing (see Section 2.3.1); another one is when it is necessary to type-erase operations that are not supported by the macro syntax (e.g. conversion operators [7]) . Finally, macros can only be defined at namespace scope, which may be a limiting factor at least for readability.

When the usage scenario is simple enough, on the other hand, Boost.TypeErasure does offer a very convenient notation. For instance, the following is all that is required in order to support the working example defined in Section 2.1 (compare this with the amount of code required by Adobe.Poly):

```
namespace erasure = boost::type_erasure;

BOOST_TYPE_ERASURE_MEMBER((has_get_area), get_area)
BOOST_TYPE_ERASURE_MEMBER((has_get_perimeter), get_perimeter)

using ShapeRequirements = boost::mpl::vector<
    erasure::copy_constructible<>,
    has_get_area<double(), erasure::_self const>,
    has_get_perimeter<double(), erasure::_self const>,
    erasure::relaxed>;

using Shape = erasure::any<ShapeRequirements>;
```

The `BOOST_TYPE_ERASURE_MEMBER` macro allows generating template code for concepts that require the presence of a member with a certain name in modeling types (e.g. `has_get_area` requires the presence of a member function called `get_area()`). For member functions, the signature is *not* specified as an argument to the `BOOST_TYPE_ERASURE_MEMBER` macro; rather, it is specified as the first template argument to the generated concept (e.g. `has_get_area<>`) when used inside a larger set of requirements (e.g. the `ShapeRequirements` compile-time sequence). The second template argument to the generated concept specifies whether the corresponding member function is `const`-qualified, in which case `_self const` should be used. When this argument is omitted, it defaults to `_self`, which denotes a mutating function.

The `copy_constructible` concept is a predefined concept provided by the library, and the `relaxed` constraint instructs the framework to use certain default settings that regulate the behavior of the library under particular circumstances – a deeper analysis is beyond the scope of this document.

The usage of compile-time MPL sequences of micro-constraints to define concepts, as well as the adoption of placeholders such as `_self`, do not help readability; however, a comparison with the amount of code required for the same use case by Adobe.Poly (see Section 2.2) makes the benefits of using Boost.TypeErasure apparent.

2.3.1. Adaptation vs. duck-typing

Let's suppose we want our `Shape` wrapper to be able to erase the class `Triangle` from Section 2.2.1. Because the names of the member functions of `Shape` do not match those of the member functions of `Triangle`, we cannot use the `BOOST_TYPE_ERASURE_MEMBER` macro as we did in the previous section. We therefore need to explicitly define one class template for each required member function:

```
template<class T>
struct has_get_area
{
    static double apply(T const& x) { return x.get_area(); }
};

template<class T>
struct has_get_perimeter
{
    static double apply(T const& x) { return x.get_perimeter(); }
};
```

Next we'll have to inform the framework about the existence of our concepts by specializing the `concept_interface` template (an explanation of the details is beyond the scope of this document):

```
namespace boost { namespace type_erasure
{
    template<typename C, typename Base>
    struct concept_interface<has_get_area<C>, Base, C> : Base
    {
        double get_area() const
        { return call(has_get_area<C>(), *this); }
    };

    template<typename C, typename Base>
    struct concept_interface<has_get_perimeter<C>, Base, C> : Base
    {
        double get_perimeter() const
        { return call(has_get_perimeter<C>(), *this); }
    };
} }
```

We will also have to rewrite our `ShapeRequirements` class, because the definition of the `has_get_area<>` and `has_get_perimeter<>` templates has changed, and the signature parameter is no longer required:

```
using ShapeRequirements = boost::mpl::vector<
    boost::type_erasure::copy_constructible<>,
    has_get_area<boost::type_erasure::_self>,
    has_get_perimeter<boost::type_erasure::_self>,
    boost::type_erasure::relaxed>;
```

This code is completely equivalent to the one we wrote in the previous section, only more verbose. However, it gives us enough flexibility to define custom adaptation logic by specializing the `has_get_area<>` and `has_get_perimeter<>` templates for the `Triangle` class:

```
template<>
struct has_get_area<Triangle>
{
    static double apply(Triangle const& x) { return x.compute_area(); }
};

template<>
struct has_get_perimeter<Triangle>
{
    static double apply(Triangle const& x) { return x.compute_perimeter(); }
};
```

This completes the adaptation work. Both readability and verbosity tend to match those of Adobe.Poly (see Section 2.2.1 for a comparison).

It is worth mentioning that things would become slightly more cumbersome if we wanted our concept templates to accept further template parameters, such as as the type of a function argument. The following example, taken from [20], shows how to specialize `concept_interface` for a generic constraint `has_push_back<T>` that requires the presence of a member function `push_back()` accepting an argument of type `T`:

```
namespace boost { namespace type_erasure
{
    template<class C, class T, class Base>
    struct concept_interface<has_push_back<C, T>, Base, C> : Base
    {
        void push_back(typename as_param<Base, T const&>::type arg)
        { call(has_push_back<C, T>(), *this, arg); }
    };
} }
```

2.3.2. Reference semantics

Unlike Adobe.Poly, Boost.TypeErasure offers built-in support for reference semantics, and taking advantage of it is very simple. The semantics of an erasing wrapper is specified when instantiating the `any` class template. In Section 2.3 above we defined our `Shape` wrapper as follows:

```
using Shape = erasure::any<ShapeRequirements>;
```

In order to turn `Shape` into a wrapper with reference semantics, it is sufficient to provide an additional template argument to `any`, like so:

```
using ShapeRef = erasure::any<ShapeRequirements, erasure::_self&>;
```

This allows us to write code similar to the one from Section 2.2.2 (notice, that using `std::ref()` is not required here):

```
int main()
{
    Rectangle r{{1.0, 2.0}, 5.0, 6.0};

    ShapeRef s{r};

    std::cout << s.get_area() << std::endl; // PRINTS 30

    r.set_size(4, 2); // SUPPOSE SUCH A FUNCTION EXISTS

    std::cout << s.get_area() << std::endl; // PRINTS 8
}
```

Notably, rvalue references are supported in a similar way. In fact, it is possible to define an erasing wrapper like so:

```
using ShapeRefRef = erasure::any<ShapeRequirements, erasure::_self&&>;
```

The `ShapeRefRef` wrapper could then be initialized as shown below:

```
auto r = Rectangle{{1.0, 2.0}, 5.0, 6.0};

auto s = Shape{r};

ShapeRefRef s1 = std::move(r); // OK

ShapeRefRef s2 = std::move(s); // OK
```

Unfortunately, like every library solution, Boost.TypeErasure is unable to solve the Proxy Dilemma [2], and therefore suffers from limitations similar to those described in Section 2.2.2. In particular, compile-time generic algorithms are not guaranteed to work transparently with erasing wrappers [19], and ownership of type-erased objects through smart pointers is limited in flexibility.

2.3.3. Conclusions on Boost.TypeErasure

Boost.TypeErasure is a powerful, modern C++ library that supports DGP with both value semantics and reference semantics. It makes simple usage scenarios relatively easy to handle by providing macros that improve readability and predefined concepts that save the user some repetitive work.

However, the syntactic convenience of Boost.TypeErasure with respect to Adobe.Poly tends to fade away when using adaptation rather than duck-typing; more generally, accomplishing non-trivial tasks through Boost.TypeErasure requires (1) being acquainted with TMP techniques and (2) being ready to interpret long, hard-to-decipher compiler errors – this may sound less problematic then it actually is.

For a demonstration of the kind of complexity involved with solving relatively simple problems, the reader is invited to refer to the documentation page showing how to define a concept which requires the presence of two different overloads of a member function [21].

Complexity aside, Boost.TypeErasure offers a lot of interesting functionality, such as support for associated types, covariant function arguments, and implicit type-safe conversions between concepts. It also has a macro `BOOST_TYPE_ERASURE_FREE` which allows defining freestanding function requirements pretty much in the same way as the `BOOST_TYPE_ERASURE_MEMBER` macro allows defining member function requirements (see Section 2.3).

The limits of Boost.TypeErasure are those inherent to all library solutions for DGP: reference semantics through smart pointers is not easy to achieve, and the Proxy Dilemma [2] cannot be solved.

Last but not least, the expressivity and flexibility of Boost.TypeErasure come at the expense of relatively high build times, due to the heavy use of TMP techniques in the library's internals.

2.4. Pyry Jahkola's Poly

The Poly Library [9], authored by Pyry Jahkola, is a modern C++ library inspired by Adobe.Poly. It is not meant for production use and lacks a few fundamental features (like support for reference semantics), so it will not be covered extensively in this document. For documentation, the reader can refer to [9].

By design, Poly does not support member function call syntax on type-erasing wrappers: instead, it forces the use of freestanding functions. Therefore, our function `print_areas()` from Section 2.1 will have to be rewritten like so:

```
void print_areas(std::vector<Shape> const& v)
{
    for (auto const& s : v)
    {
        std::cout << "Area: " << get_area(s) << std::endl;
    }
}
```

The first step consists in defining the interface of our type-erased wrapper. This is easily done by (1) using the `POLY_CALLABLE()` macro to generate callable objects with the names of the functions required by our `Shape` concept, and (2) instantiating the `poly::interface<>` variadic class template with a corresponding list of function signatures:

```
#include <poly/interface.hpp>

POLY_CALLABLE(get_area);
POLY_CALLABLE(get_perimeter);

using Shape = poly::interface<
    double(get_area_, poly::self const&),
    double(get_perimeter_, poly::self const&)>;
```

The first argument type of each concept function signature should be the name of the corresponding callable defined through the `POLY_CALLABLE()` macro, with an underscore appended.

Then, we shall define one function template named `call()` – which will be recognized by the framework – for each function required by our concept; this implements the dispatch mechanism:

```
template<typename T>
double call(get_area_, T const& x)
{
    return x.get_area();
}

template<typename T>
double call(get_perimeter_, T const& x)
{
    return x.get_perimeter();
}
```

This is everything that needs to be in place for realizing the use case from Section 2.1. When adaptation is necessary, e.g for type-erasing the `Triangle` class from Section 2.2.1, proper (non-template) overloads of the `call()` function templates must be defined to specialize the dispatch:

```
double call(get_area_, Triangle const& x)
{
    return x.compute_area();
}

double call(get_perimeter_, Triangle const& x)
{
    return x.compute_perimeter();
}
```

The way Poly enables DGP in C++ is elegant and simple, but the library lacks a few important features. In particular, as already mentioned, it does not support reference semantics by design – and therefore does not attempt to address the Proxy Dilemma [2]; the design choice of forcing the use of freestanding functions on type-erasing interfaces also limits its generality to some extent.

2.5. Boost.Interfaces

Jonathan Turkanis's Boost.Interfaces library [10] (which, in spite of its name, is not related to the Boost C++ Libraries) is part of Christopher Diggins's Object-Oriented Template Library (OOTL) library [22]. Boost.Interfaces is one of the first open-source projects that historically tackled the problem of supporting DGP in C++. Unfortunately, both Boost.Interfaces and the OOTL library seem to be no longer maintained, and their obsolete dependencies made it impossible for us to evaluate them. Besides, Boost.Interfaces apparently never evolved from an experimental stage into a real-world product ready for production use.

Based on the documentation available on the project's website, Boost.Interfaces allows creating custom type-erasing wrappers by specifying the interface that erased types have to adhere to by using macros. Whether a type conforms to an erasing wrapper's interface is determined exclusively through duck-typing: Boost.Interfaces does not support adaptation.

Boost.Interfaces's strong point is simplicity. For instance, this is how the type-erasing `Shape` wrapper for our working example from Section 2.1 would be defined:

```
BOOST_IDL_BEGIN(Shape)
    BOOST_IDL_FN0(get_area, double)
    BOOST_IDL_FN0(get_perimeter, double)
BOOST_IDL_END(Shape)
```

There are several downsides to this approach: in particular, the requirements for movability and copyability cannot be expressed through this simple mechanism. Perhaps this is one of the reasons why Boost.Interfaces only supports reference semantics.

To allow unique and shared ownership of type-erased objects, Boost.Interfaces provides its own smart pointers, which are meant to emulate the behavior of standard smart pointers – for more information, see [23].

2.6. Signatures

Unlike all the projects mentioned so far, Signatures [12] attempts to introduce support for DGP in C++ at a core language level. An implementation of Signatures as a language extension was available in GCC up to version 2.95 [13] [14].

The key notion introduced by Signatures in the C++ language is – unsurprisingly – that of a *signature*. A signature defines an interface that modeling types have to conform to in order to be treated polymorphically through a signature pointer or signature reference. The mechanism is non-intrusive: modeling types are not required (in fact, they are not allowed) to derive from a signature.

Like Boost.Interfaces (see Section 2.5), Signatures only focuses on reference semantics, and does not allow using type-erased wrappers with value semantics. Therefore, the original working example from Section 2.1 will have to be rewritten.

This is how the reworked `print_areas()` function would look like:

```
void print_areas(std::vector<Shape*> const& v) // VECTOR OF POINTERS!
{
    for (auto const s : v)
    {
        std::cout << s->get_area() << " ";
    }
}
```

The `main()` function also needs to be modified accordingly:

```
int main()
{
    auto r = Rectangle{{1.0, 2.0}, 5.0, 6.0};

    auto c = Circle{{0.0, 0.0}, 42.0};

    std::vector<Shape*> v{&r, &c}; // VECTOR OF POINTERS!

    print_areas(v);
}
```

Finally, the type-erasing wrapper `Shape` would be defined like so:

```
signature Shape
{
    double get_area();
    double get_perimeter();
};
```

Signature functions cannot be `const`-qualified, yet they are capable of delegating to types that have `const`-qualified versions of those functions – this is something I will get back to in Sections 4 and 5.

A quick comparison with the previous sections reveals the simplicity and elegance of this approach: four lines of readable code involving no template meta-programming and no macros is everything which is needed to realize our use case, at least when duck-typing is sufficient.

Signatures offers support for adaptation through so-called *views*, but the syntax is largely sub-optimal. In particular, the adaptation logic for a given type T to a given signature S is not defined once and reused wherever necessary; instead, it must be specified every time an expression of type T (resp. T*) is used to initialize a variable of type S& (resp. S*):

```
Triangle t{{0.0, 0.0}, {1.0, 0.0}, {0.0, 1.0}};

S* p = (S*; get_area = compute_area, get_perimeter = compute_perimeter)&t;
```

Perhaps the most interesting aspect of Signatures is its capability of solving the Proxy Dilemma [2]: thanks to special compiler support, generic algorithms are guaranteed to work transparently with (pointers and references to) signatures just like they do with (pointers and references to) objects that adhere to those signatures. Moreover, since signature pointers are treated just like any other pointer type at a language level, ownership through smart pointers would be possible without introducing dedicated wrappers and without modifying the existing smart pointer types.

The implementation technique adopted by Signatures [14] is the same as the one proposed for implementing virtual concepts in Section 12 of this document – at least for what concerns reference semantics.

2.7. Conclusions

In this section I have analyzed five existing solutions for DGP in C++. Two of them, namely Adobe.Poly [8] and Boost.TypeErasure [5], are robust, well-known open-source libraries ready to be used in a production environment. The former is mostly focused on supporting value semantics and requires a certain amount of boilerplate code to be written by the user; the latter supports both value semantics and reference semantics, and tends to require less work for simple use cases. However, it heavily relies on TMP techniques, which slows down build time, and tends to become unwieldy for more complex usage scenarios. Both Adobe.Poly and Boost.TypeErasure support duck-typing as well as adaptation.

I also discussed Pyry Jahkola's Poly library [9], which is not ready for production use at the moment of writing, but provides a very convenient syntax for generating type-erasing wrappers. By design it only allows using freestanding functions on concept interfaces, and does not support reference semantics.

The Boost.Interfaces project [10] (part of the OOTL library [22]) is probably the earliest attempt to address the problem of DGP in C++ at a library level, and seems to be no longer under active development. Boost.Interfaces is only concerned with reference semantics, and allows defining type-erasing wrappers by using preprocessor macros. Adaptation is not supported.

None of the library solutions mentioned so far are capable of solving the Proxy Dilemma [2], which limits the possibility of using transparently type-erasing wrappers and the types they erase in generic algorithms; another consequence of creating wrapper proxies is that existing smart pointers are not suitable for owning type-erased objects, and dedicated smart pointers need to be provided instead; finally, supporting proper overload resolution in the presence of type-erasing wrappers is likely to require complicated user-defined conversions.

The Signatures project [12] tries to address the problem at a core language level by introducing a new language construct (*signatures*) and dedicated compiler support. Signatures was implemented as a language extension in GCC up to version 2.95 [13]. It only supports reference semantics, but does solve the Proxy Dilemma and therefore is not affected by the issues mentioned in the previous paragraph.

All the solutions currently available today for C++ programmers to realize DGP have limitations either in terms of ease of use or compile-time overhead; another factor which significantly limits the usability of existing tools is the poor quality of compiler diagnostics.

Even setting these issues aside, the amount of boilerplate code the user is required to write and the number of details that needs to be taken care of is still needlessly higher than the inherent complexity of the problem at hand. An interesting approach to reducing this burden through the use of automation tools is presented in [24]. A better way to achieve the same result, provided C++ will offer support for reflection at some point, would be to use this mechanism for generating at least part of the boilerplate code at compile-time. Even in that case, however, library solutions will not be able to solve the Proxy Dilemma nor the issue of high build times.

3. Design goals

This section states the fundamental goals that our approach to DGP pursuits and describes their impact on the design of the solution I propose. In approximate order of importance, our goals are:

- Simplicity and usability
- Support for value semantics and reference semantics
- No proxy objects
- Efficiency
- Integration with related language and library features
- Fast build times

The following sections will expand on each of these points.

3.1. Simplicity and usability

Type-erasure is, in principle, a simple idea. Existing library solutions require the user to write too much boilerplate code, especially for realizing non-trivial use cases. Even when the amount of work needed is not considered a limiting factor, the result tends to be code which is hard to read and/or maintain. Maintenance is especially problematic due to the very low quality of compiler diagnostics, which can easily consist of pages of template instantiation stack traces.

Endorsing Bjarne Stroustrup's mission of "*Making Simple Tasks Simple*" [25], our goal is to provide users with tools that (1) do not require them to deal with TMP unless parameterization is part of their *domain* problem, (2) do not require them to use macros in order to improve readability, and (3) do not require them to write boilerplate code and/or perform repetitive tasks in order to realize their use cases.

In our opinion, language support is a *condicio sine qua non* to achieve these goals, and the Signatures project [12] shows the potentials of a language-level solution for DGP in terms of simplicity and flexibility – compare, for instance, the code from Section 2.6 with the snippets from Sections 2.2-2.5.

3.2. Support for value semantics and reference semantics

Value semantics is a popular programming style in the C++ community [26], and the C++ language encourages the definition of types that "behave like an `int`". The benefits of value semantics and the need for a non-intrusive form of polymorphism that supports them are well explained in [16], [18], and [27]. Supporting value semantics with DGP is therefore a crucial aspect of our solution.

However, it is sometimes necessary or convenient to adopt reference semantics for solving certain problems. The C++ language supports reference semantics through polymorphism based on inheritance and pointers/references; unfortunately, as an intrusive mechanism, inheritance causes the series of problems discussed in Section 1.

It is therefore important to provide support for DGP with reference semantics as well, and to do so in a way that integrates seamlessly with all the elements of modern C++ programming, including smart pointers and rvalue references.

3.3. No proxy objects

One of the problems that affect library solutions supporting reference semantics is the so-called Proxy Dilemma [2]. In order to access objects through an interface different from the one naturally defined by their types, pointer-like proxy objects must be created that perform the necessary adaptation.

These proxy objects are separate objects from the ones they are meant to abstract. In other words, supposing that Shape is some sort of type-erasing wrapper akin to the ones introduced in Section 2, the assertion in the code below will always fire:

```
auto r = Rectangle{{1.0, 2.0}, 5.0, 6.0};

auto s = ShapeRef{r};

assert(&r == &s); // THIS ASSERTION ALWAYS FAILS
```

This has a number of implications: firstly, generic algorithms might not work transparently with wrapper types as they do with the types erased by those wrappers [19]; secondly, different kinds of wrappers are necessary to support the different kind of references that exist in C++ (namely, *lvalue* and *rvalue* references); thirdly, reference-like wrappers cannot be used just like C++ references, since the language forbids overloading the dot operator for accessing class members; finally, ownership of type-erased objects with dynamic storage duration through smart pointers would require introducing dedicated smart pointer classes or specializing existing ones. All of these issues introduce unnecessary complexity (see Section 3.1) and/or limit expressivity (see Section 3.2).

The requirement is for the following to be valid C++ code, and the assertion in the last line to always succeed:

```
auto r = Rectangle{{1.0, 2.0}, 5.0, 6.0};

Shape& s = r;

assert(&r == &s); // THIS ASSERTION SHOULD ALWAYS SUCCEED
```

3.4. Efficiency

Efficiency is a primary concern for C++ developers as well as for the designers of the language. Our proposed solution aims at supporting DGP with no performance penalty with respect to the idiomatic solutions currently in use while guaranteeing a performance improvement in most situations.

Concerning reference semantics, the solution I propose tends to be superior to inheritance-based polymorphism because types are not required to be polymorphic (i.e. to have at least one virtual function) in order to be accessed through type-erasing virtual concepts; more to the point, this means that the instances of those types do not need to carry a *vtable* pointer (or several such pointers) regardless of whether they will be used polymorphically or not. This minimizes memory consumption.[1]

When objects are accessed polymorphically through virtual concepts, the dispatch mechanism is no different from the one that realizes a regular virtual function call; this means that our solution is performance-wise no worse than inheritance-based polymorphism.

Using heterogeneous types with value semantics polymorphically is not allowed at a language level, so the term of comparison for our proposal in this respect are the library solutions described in Sections 2.2-2.4.

The main technical obstacle with supporting value semantics is that, in general, the size of an erased object is unknown at compile-time. The consequence is that although an object which is used with value semantics has automatic storage duration (because its lifetime ends when execution leaves the block it is declared in), it cannot be allocated on the stack. Instead, the storage which holds the object must be allocated from the free store.

Dynamic memory allocation is an expensive procedure which might be problematic or unaffordable in certain scenarios. The performance penalty can be alleviated by adopting optimization techniques such as SBO and/or by providing custom allocation mechanisms. While SBO is covered in Section 12, the current version of this proposal is not concerned with customizing the allocation process. Existing library solutions do not offer this flexibility either.

In any case, the performance implications of dynamic allocations will only be relevant when using types with value semantics polymorphically: other use cases (e.g. polymorphic access of objects with reference semantics) are not affected. This is in line with the well-known motto "*you don't pay for what you don't use*", which is also one of C++'s fundamental design principles.

[1] The most natural implementation of concept pointers and concept references is as a pair of regular pointers (see Section 12 for the details). In situations where the number of pointers or references largely exceeds the number of objects, this solution will lead to higher memory consumption. Moreover, if concept pointers are copied around frequently and/or in performance-critical portions of an application, a penalty in terms of speed may also be observed. However, invoking functions through a concept pointer might be a more cache-friendly operation than invoking functions through a regular virtual table, because it would not be necessary to fetch the object from main memory in order to access its *vptr*.

3.5. Integration with related language and library features

Introducing language support for DGP obviously calls for dedicated syntax in which to express the interface of type-erasing wrappers. One of the aims of this proposal is to integrate seamlessly with related language features, rather than clashing with them, on the syntactic (and semantic, where possible) plane.

Hence, our proposal allows reusing concepts as defined by the Concepts TS [6] with only a few limitations. In particular, the `Shape` wrapper from our working example in Section 2.1 would be automatically generated by the compiler from the following concept definition:

```
template<typename T>
concept bool Shape = requires(T const& x)
{
    { x.get_area() } -> double;
    { x.get_perimeter() } -> double;
};
```

In terms of clarity this is close to the syntax offered by Signatures [12], but more expressive – as I will show in Section 4 and 5. The dynamic usage of a concept like `Shape` would be disambiguated from its static usage through the `virtual` keyword. For instance:

```
void print_areas(std::vector<virtual Shape*> const& v) // NOT A TEMPLATE!
{
    for (auto const& s : v)
    {
        std::cout << s->get_area() << " ";
    }
}
```

An alternative way of writing the above would be to qualify the concept as `virtual` in its definition. This would make it unnecessary to explicitly qualify its usages:

```
template<typename T>
virtual concept bool Shape = requires(T const& x)
{
    { x.get_area() } -> double;
    { x.get_perimeter() } -> double;
};

void print_areas(std::vector<Shape*> const& v) // NOT A TEMPLATE!
{
    for (auto const& s : v)
    {
        std::cout << s->get_area() << " ";
    }
}
```

Similar considerations, which have been partly covered in Section 3.3, can be made for the integration of virtual concepts with C++'s type system. In particular, forming concept pointers and references should be possible with the same syntax and semantics used for regular compound types.

Given the last definition of Shape given above, for example, the following initializations should all be legal and have the obvious meaning, and no assertions should be triggered:

```
auto r = Rectangle{{1.0, 2.0}, 5.0, 6.0};

Shape& s = r;

assert(&r == &s); // SHALL NEVER FIRE

std::unique_ptr<Shape> up = std::make_unique<Rectangle>(r);

assert(up.get() != r); // SHALL NEVER FIRE

Shape const* p = up.get();

std::shared_ptr<Shape> sp = std::move(up);

assert(sp.get() == p); // SHALL NEVER FIRE
```

Extensions to existing language features are sometimes needed to cover all the relevant use cases for type erasure, e.g. adaptation; while C++0x's *concept maps* [28] (i.e. adaptation in the context of static polymorphism) were dropped from the scope of the Concepts TS [6], adaptation is a common use case for type erasure, which is why I chose to propose a syntax to support it.

However, since I also chose to establish (where possible) an isomorphism between virtual concepts and compile-time concepts as defined in Concepts TS, this raises the question whether it would not be wise for this feature to be developed in parallel on both shores of the river – i.e. for DGP and classical GP with templates. I chose to provide some syntax to express adaptation, while remaining open to the possibility of dropping this feature for the sake of consistency and better integration with Concepts TS if deemed necessary.

Interaction with other language features such as inheritance, templates, and overload resolution ought to be carefully considered and will be discussed in detail throughout this document (in particular, see Sections 9 and 10).

3.6. Fast build times

One of the greatest benefits of inheritance-based polymorphism is that it enables a form of *dependency inversion* [3] thanks to which a software component can transfer execution to another software component at run-time without depending on its definition at compile-time.

Thanks to the layer of indirection provided by virtual functions and inheritance, modifying the definition of any type which implements an abstract interface does not require recompiling any of the clients using that interface. Apart from enabling independent deployment, inheritance and virtual functions represent a compilation firewall that helps keeping build times manageable in spite of C++'s sub-optimal compilation model.

One of our concerns with existing library solutions for DGP in C++ is that they tend to rely heavily on TMP techniques in order to save the user from dealing with repetitive tasks; this, unfortunately, leads to a dramatic increase in build times, making the benefits of a compilation firewall less perceivable.

One of the goals of our solution is therefore to support DGP with a compilation overhead comparable to the one implied by traditional, inheritance-based polymorphism; in particular, virtual concepts should lead to build times significantly faster than those obtained with the library solutions examined in Sections 2.2-2.5.

4. Virtual concepts in a nutshell

This section is meant to provide a quick overview of the capabilities and use cases of virtual concepts. Each subsection in the following discussion will focus on one relevant aspect of this proposal; because each topic will be covered more extensively in later sections of this document, I aim to give the presentation a more pragmatic slant here. In particular, implementation details and theoretical aspects are outside the scope of this section.

4.1. Virtual usage of concepts and syntactic symmetry

One important clarification that ought to be made at this point is that what I called "virtuality" of a concept refers to its *usage*, not its essence. Concepts *per se* are just sets of requirements that specify what can be done with a certain class of types; whether these contract definitions are used in the context of static, template-based polymorphism rather than in the context of dynamic, erasure-based polymorphism does not change much about their nature – except for the fact that the dynamic usage of concepts imposes a few limitations on the kind of requirements that can be expressed.

To clarify things with an example, let's consider once again the concept Shape from Section 3.5:

```
template<typename T>
concept bool Shape = requires(T const& x)
{
    { x.get_area() } -> double;
    { x.get_perimeter() } -> double;
};
```

This concept definition is compatible with the syntax of Concept TS [6], and can be used in a function template like the following:

```
template<typename T>
    requires Shape<T>
bool is_big(T const& obj)
{
    return obj.get_area() > 10.0;
};
```

The terse syntax introduced by Concepts TS also allows rewriting the above like so:

```
bool is_big(Shape const& obj)
{
    return obj.get_area() > 10.0;
};
```

In spite of the terse notation, `is_big()` is a function *template*: when called with an argument of type `Rectangle`, `is_big()` will get instantiated for that type, and inside the function's body the type of `obj` will be known to be exactly `Rectangle`.

In order for this to be possible, the definition of `is_big()` must be visible at the call site, and cannot be hidden in an independently-compiled translation unit.

However, the same concept `Shape` can be used for dynamic polymorphism to erase all types that satisfy the corresponding requirements. This is what I would call a *virtual usage* of `Shape`. Virtual usage of a concept is discriminated by the presence of the `virtual` keyword before the parameter declaration:

```
bool is_big(virtual Shape const& obj)
{
    return obj.get_area() > 10.0;
};
```

Here, `is_big()` is *not* a function template and can be compiled separately from the client code; however, inside of its body the concrete type of `obj` is unknown – which is likely to prevent certain compiler optimizations.

The role of the `Shape` concept in the two scenarios is quite similar: in both cases the concept allows verifying at compile-time that a generic algorithm will only be invoked with types that satisfy its requirements.

I decided to reflect this symmetry between the static and dynamic usage of concepts on the syntactic plane too; in particular, since Concepts TS introduces the possibility of using terse syntax for function templates, I introduced the possibility of using template syntax for regular functions:

```
template<typename T>
    requires virtual Shape<T>
bool is_big(T const& obj)
{
    return obj.get_area() > 10.0;
};
```

There is a more practical reason for allowing this than just symmetry. The reason becomes clear when considering a function whose parameter list contains more than one occurrence of the same concept name. For instance:

```
bool is_smaller_than(virtual Shape const& lhs, virtual Shape const& rhs)
{
    return lhs.get_area() < rhs.get_area();
};
```

In the above code, should `lhs` and `rhs` be of the same modeling type (i.e. both `Rectangle`s, both `Circle`s, and so on), or could they have different types – as long as they both model the `Shape` concept?

Consistently with the design of Concepts TS, repeated occurrences of the same concept name in the same function definition denote identical types. In other words, the function call below would be illegal:

```
Rectangle r{{0.0, 0.0}, 1.0, 5.0};

Circle c{{1.0, 2.0}, 3.0};

std::cout << is_smaller_than(r, c); // ERROR: NOT THE SAME TYPE!
```

This may or may not be desirable, depending on the situation. If it is not, the template syntax allows us to trade some verbosity for some flexibility:

```
template<typename T, typename U>
    requires virtual Shape<T> && virtual Shape<U>
bool is_smaller_than(T const& lhs, U const& rhs)
{
    return lhs.get_area() < rhs.get_area();
};
```

Notice, that `is_smaller_than()` is *not* a function template here: its definition can be safely put in a separately-compiled translation unit.

The fact that template syntax is being used with non-template semantics may be misleading (just like non-template syntax used with template semantics is), yet it is convenient if we want to be able to use regular concept definitions (i.e. with syntax and semantics of Concepts TS) and turn them into virtual concepts on-demand.

Choosing a different notation, while possible, would have likely resulted in duplication and overlap. Disambiguation through the `virtual` keyword, on the other hand, reflects the ontological similarity between compile-time and run-time concepts.

In certain situations it might be helpful to toggle the "*virtual by default*" semantics, so that using explicit annotation is not required at every virtual usage of a concept. This can be done by prefixing the concept's definition with the `virtual` keyword:

```
template<typename T>
virtual concept bool Shape = requires(T const& x)
{
    { x.get_area() } -> double;
    { x.get_perimeter() } -> double;
};

// NOT A FUNCTION TEMPLATE!
// THE Shape CONCEPT IS USED VIRTUALLY HERE
bool is_big(Shape const& obj)
{
    return obj.get_area() > 10.0;
};
```

When a concept's definition is marked as `virtual`, non-virtual usage of that concept in a function's parameter list can be expressed through the `static` keyword. The version of `is_smaller_than()` shown below defines a function *template*:

```
// THIS IS A FUNCTION TEMPLATE!
bool is_smaller_than(static Shape const& lhs, static Shape const& rhs)
{
    return lhs.get_area() < rhs.get_area();
};
```

In spite of the syntactic uniformity, there are a few situations where virtual concepts lose some of the flexibility offered by their static counterpart: for example, concepts that define type requirements depending on the modeling type variable lead to degenerate erasure unless additional constraints are provided – see Sections 4.2, 4.6 and 5.

4.2. Degenerate erasure and `std::any`

In order to better characterize the nature of virtual concepts and their relationship with classical compile-time concepts, it is beneficial to examine their most degenerate form: the `Void` concept.

The definition of `Void` is trivial in that it always evaluates to `true` for any type, and places no requirements on its models:

```
template<typename T>
virtual concept bool Void = true;
```

This fact has two consequences: the first consequence is that a generic algorithm accepting a `Void` parameter could be invoked with *any* argument; the second consequence is that no specification whatsoever is given of what the generic algorithm can *do* with an object erased by a `Void` value. Because of this, the only sensible operation a generic algorithm could perform on a `Void` concept used virtually is downcasting it to its original type.[2] Specifically:

```
void foo(Void& x)
{
    auto i = dynamic_cast<int>(x); // THROWS std::bad_cast ON FAILURE

    // ...
}
```

Downcasting an object of concept type is possible by using `dynamic_cast` or `static_cast`, with the usual semantics (see Section 11 for more information). Notably, the introduction of language support for DGP allows reusing familiar operators without requiring library additions specifically tailored for one single concept, such as `std::any_cast<>` for `std::any`.

[2] Notice that parameters of type `Void` cannot be taken by value, because that would require at least movability, whereas `Void` places no requirement on its modeling types.

The `std::any` class (inspired by its Boost counterpart, Boost.Any [29]) is an interesting library addition that is likely going to be part of the next C++ Standard. Contrary to what the name suggests, `std::any` is not meant to erase objects of *any* type – which is what `Void` would do. Rather, it expects compatible types to support certain common operations (including copy-construction and no-throw move-construction), but no more than that.

By using concepts, the requirements of `std::any` could be expressed this way:

```
template<typename T>
concept bool Any = requires(T x)
{
    requires DefaultConstructible<T> &&
             CopyConstructible<T> &&
             NoThrowMoveConstructible<T> &&
             CopyAssignable<T> &&
             NoThrowMoveAssignable<T> &&
             Destructible<T>;
};
```

This definition would allow us to write code like the following:

```
std::vector<virtual Any> v;

v.push_back("Hello"s);
v.push_back(42);

std::vector<int> integers{1, 2, 3};
v.push_back(integers);

assert(dynamic_cast<int>(v[1]) == 42);
```

It is easy to see that the definition of Any given above is not only self-documenting, but also significantly simpler than its library counterpart `std::any`; moreover, it supports both value semantics and reference semantics, and is likely to result in faster build times due to the absence of template code.

4.3. Expression requirements and signature requirements

The earliest proposals for adding compile-time concept support to the C++ language allowed expressing requirements in terms of function *signatures* that modeling types had to define.

Later on, this design choice was discarded because it did not offer a natural way of defining concepts such as `MoveConstructible`: in order for a type to be move-constructible, the presence of a move-constructor is a *sufficient* condition, but not a necessary one. Since move operations can decay to copies, a better mechanism than lists of signatures was needed for requiring “either the presence of a move-constructor or of a copy-constructor”.

As it turns out, expressing this requirement for a certain type T is simple if we are allowed to mandate that, for some invented object x of type T, the *expression* `T{std::move(x)}` should be well-formed. After all, this is how the Standard formalizes its own concepts.[3]

Because of this, Concepts TS [6] allows parameterized expressions to be used for defining a concept's requirements; these expressions must be well-formed when instantiated for a given type. For example:

```
template<typename T>
concept bool EqualityComparable = requires(T x)
{
    { x == x } -> bool;
};
```

The above notation is compact and convenient, but it has a subtle drawback. Consider the following type:

```
struct Bogus
{
    friend auto operator == (Bogus&, Bogus&) -> bool;
};
```

This type does model `EqualityComparable` as we defined it, because all of the expressions in the definition of `EqualityComparable` are well-formed when the type variable T is replaced with `Bogus`: x is an *lvalue*, and lvalue references can bind to lvalues.

However, it is easy to see that a comparison between two *rvalues*, or even an rvalue with an lvalue (or vice versa), is incompatible with this definition of `EqualityComparable`, because its modeling types are not *required* to have an equality-comparison operator that is capable of accepting rvalue arguments.

Interestingly, this is not a fatal issue for static concepts. To see why, consider the following code:

```
T lexical_cast(std::string const& s)
{
    /* ... */
}

template<typename T>
    requires EqualityComparable<T>
void something(T a)
{
    auto s = /* GET STRING FROM USER */;
    if (a == lexical_cast<T>(s))
    {
        // ...
    }
}
```

[3] Interesting insights as well as a historical overview of the problem are given in [36].

Per Concepts TS, the definition of `something()` above is legal, even though the right-hand side argument of the equality-comparison is an rvalue: after all, the compiler will *not* check the body of a function template against the definition of the concepts that constrain its parameters' types.[4]

When invoking `something()` as in the snippet below, on the other hand, a regular instantiation error will be generated by the compiler:

```
int main()
{
    Bogus b;

    something(b); // INSTANTIATION ERROR FROM INSIDE something()'S BODY!
}
```

In other words, with compile-time concepts we would have nice, early diagnostics when instantiating a generic algorithm with types that do not satisfy the *minimum* requirements placed by the corresponding concept definitions; on the other hand, when instantiating generic algorithms with types that do satisfy the minimum requirements of some under-specified concepts, but are not compliant with the *usage* which is made of those types inside the generic algorithm, a classical, late diagnostic would be emitted. All in all, types which are inappropriate for the algorithm cause compilation errors even in the presence of under-specified concepts.

With virtual concepts, however, things are different. Since generic algorithms can be compiled separately and do not know what concrete types they will be instantiated with, the compiler must validate the algorithm's body against the concept's definition: if the algorithm is trying to perform operations that do not correspond to any of the allowed expressions, compilation fails. Clearly, under-specified concepts are a serious problem in this context.

If we were to provide a more complete definition of the `EqualityComparable` concept above, this is what it would look like:

```
template<typename T>
concept bool EqualityComparable = requires(T x)
{
    { x == x } -> bool;
    { x == std::move(x) } -> bool;
    { std::move(x) == x } -> bool;
    { std::move(x) == std::move(x) } -> bool;
};
```

The scalability issue with this approach is evident: allowing the users of a concept to pass both lvalues and rvalues for all the N arguments of a function call will require specifying 2^N requirements.

[4] A recent presentation by prof. Andrew Sutton at CppCon 2014 [37] seems to indicate that this might be the case in the future. If that were true, the same issue I are pointing out in this section for run-time concepts would become relevant for compile-time concepts as well, especially considering that most of the concept definitions published so far, including those in the Origin library [38], only use lvalue expressions in their requirements.

While this particular example could be rescued by declaring x to have type T `const` rather than T, the reason may not be obvious, and in the general case doing so may not be appropriate or possible (e.g. if we wanted rvalues to be moved-from).

To avoid the combinatorial explosion of requirements, I advocate the need for an alternative way of expressing them. As it turns out, signature-based requirements in their original formulation allow for the desired kind of clarity and flexibility:

```
template<typename T>
concept bool EqualityComparable = requires()
{
    auto operator == (T const&, T const&) -> bool;
};
```

To be fair, there is one problem with this solution too: the equality-comparison operator may be defined as a free function or as a member function, and requiring only one would give us (again) an under-specified concept.

This is, however, a less serious problem to deal with, for two reasons: first, the kind of functions which could be defined either as members or as freestanding functions is limited by the language; second, these functions never have more than two arguments, which reduces the impact of the combinatorial explosion of requirements. Hence, when we want to specify that all value categories are accepted for all of a function's parameters, we can afford using expression requirements for concepts such as `EqualityComparable`, while resorting to signature requirements in the general case:

```
template<typename T, typename U, typename V>
virtual concept bool Fooable = requires()
{
    auto foo(T x, U y, V z) -> void; // ONE REQUIREMENT INSTEAD OF EIGHT!
};
```

In general, a function signature requirement is fulfilled by a certain type if and only if a forwarding implementation of that function written in terms of the `INVOKE()` machinery (as defined by the C++ Standard in paragraph 20.9.2 [func.require]) would be well-formed.

More concretely, given a signature requirement S, we translate the problem of deciding whether a function F for type T satisfies it into the problem of deciding whether an invented variable of type `std::function<S>` can be initialized with a callable object with the same signature as F.

As an example, consider the following concept definition:

```
template<typename T>
virtual concept bool Barrable = requires()
{
    auto Barrable::bar(int x) -> bool;
};
```

We want to determine whether the following type models it or not:

```
struct Test
{
    auto bar(int&) { return true; }
};
```

To figure it out, we determine whether the following initialization is valid:

```
std::function<bool(int)> f = [] (int&) -> bool { return true; } // ERROR!
```

As it turns out, this initialization is not valid, because the internals of `std::function` will forward the argument as an rvalue, and modifiable lvalue references cannot be bound to rvalues. Hence, `Test` does not model `Barrable`.

Specifying that arguments are taken by value in function signatures helps solving the problem with the combinatorial explosion of requirements, but only if the type of the corresponding parameter is movable. If we do not want to enforce this constraint, yet we want to allow clients to provide all possible combinations of rvalues and lvalues for a function's arguments, we will need to write down 2^N requirements. If this is unacceptable, introducing special syntactic sugar is the only way out.

Since signature requirements cannot be turned into perfect-forwarding templates (see Section 5), one possibility would be to introduce a special &&& symbol that would tell the compiler to generate overloads taking the corresponding parameter, respectively, by lvalue reference and by rvalue reference. In other words, given a definition like the one below:

```
template<typename T>
virtual concept bool Barrable = requires()
{
    auto Barrable::bar(X&&& x) -> bool;
};
```

The compiler would internally rewrite it into the following definition:

```
template<typename T>
virtual concept bool Barrable = requires()
{
    auto Barrable::bar(X& x) -> bool;
    auto Barrable::bar(X&& x) -> bool;
};
```

This generalizes straightforwardly to the case of N function arguments. Clearly, introducing a new kind of reference just for this purpose – even though as syntactic sugar only – is not an elegant approach. On the other hand, the trade-off to be made is between elegance and expressiveness: whether or not this syntax should be adopted is a marginal matter for the purposes of the present proposal. After all, in its current state, the Concepts TS does not solve this problem either.

4.4. Predicates, callables, and `std::function`

As I have shown in Section 4.2, virtual concepts allow for a simple way of generating a type-erasing wrapper with the semantics of `std::any`, but without requiring a tailored implementation; something similar can be done for `std::function`.

For starters, let's define the `Predicate` concept, which models a callable object accepting one argument and returning an object of some type convertible to a `bool`:

```
template<typename F, typename T>
virtual concept bool Predicate = requires(F f, T const& x)
{
    { f(x) } -> bool;
    requires CopyConstructible<F> && Destructible<F>;
};
```

The requirement of copy-constructability and destructibility allow using the objects erased by `Predicate` with value semantics. The expression requirement makes sure that modeling types support the function call syntax, and that they can be invoked with an unmodifiable lvalue or rvalue expression as the only argument. Consistently with Concepts TS [6], virtual concepts can be parameterized by template arguments other than the one representing the modeling type variable. In this case, the parameter T specifies the type of the predicate's argument.

Without virtual concepts, similar functionality could be achieved through the following alias template based on `std::function`:

```
template<typename T>
using Predicate = std::function<bool(T const&)>;
```

The above definition looks simple enough, but the implementation of `std::function` itself is quite articulate. Here is how we could generalize `Predicate` into a `Callable` concept that can be instantiated for any signature:

```
template<typename F, typename Ret, typename... Args>
virtual concept bool Callable = requires()
{
    auto F::operator () (Args...) -> Ret;
    requires CopyConstructible<F> && Destructible<F>;
};
```

One thing worth noticing is that the above definition of the `Callable` concept *in itself* would not work as a drop-in replacement for `std::function`, because `std::function` is capable of holding an empty value, and invoking such an empty function wrapper will result in an exception of type `std::bad_function_call` being thrown. Here, initializing a `Callable` object with the `nullptr` value is not even allowed, because `std::nullptr_t` does not have a call operator.

This problem can be partially solved through *adaptation*, as discussed in Sections 4.7 and 7: adaptation allows specializing the `Callable` concept for the `std::nullptr_t` type, so that an implementation of the call operator for that type can be provided non-intrusively.

Notice that not even a nullable `Callable` concept is completely equivalent to `std::function`, because it does not require default-constructability (which would rule out lambda functions). In fact, `std::function` is more akin to a `Callable` object wrapped into an `std::optional`.

For most purposes, however, `Callable` solves the same problems as `std::function`, has a trivial definition which makes it easy to understand and teach, and demonstrates the flexibility and expressive power of virtual concepts.

4.5. Free functions, static functions, data members

So far we have mostly seen the definition of concepts that require the presence of some member function in their modeling types – except perhaps for the `EqualityComparable` concept from Section 4.3. Nevertheless, requirements can be placed on the existence of freestanding functions, static functions, and data members as well.

To set an example, this is how we defined the `Shape` concept back in Section 4.1:

```
template<typename T>
concept bool Shape = requires(T const& x)
{
    { x.get_area() } -> double;
    { x.get_perimeter() } -> double;
};
```

Had we decided to make `get_area()` and `get_perimeter()` freestanding functions, we would have written the above definition as follows:

```
template<typename T>
concept bool Shape = requires(T const& x)
{
    { get_area(x) } -> double;
    { get_perimeter(x) } -> double;
};
```

When using signature requirements instead of expression requirements, the existence of member functions `get_area()` and `get_perimeter()` can be mandated using the following syntax:

```
template<typename T>
virtual concept bool Shape = requires()
{
    auto T::get_area() const -> double;
    auto T::get_perimeter() const -> double;
};
```

Signature requirements for the existence of freestanding functions, on the other hand, look like regular function declarations:

```
template<typename T>
virtual concept bool Shape = requires()
{
    auto get_area(T const&) -> double;
    auto get_perimeter(T const&) -> double;
};
```

Static functions can also be used to define constraints, both through expression requirements and signature requirements. Here is an example with the former:

```
template<typename T>
concept bool Shape = requires(T const& x)
{
    { T::get_shape_name() } -> std::string;
    // ...
};
```

And here is an equivalent definition using signature requirements:

```
template<typename T>
concept bool Shape = requires(T const& x)
{
    static auto T::get_shape_name() -> std::string;
    // ...
};
```

Another kind of requirement that can be formulated using expression requirements is the existence of *data members* of a certain type and with a certain name. For example:

```
template<typename T>
virtual concept bool Shape = requires(T x)
{
    { x.color } -> Color;
    // ...
};
```

The `Shape` concept as defined above is only modeled by types that have a data member named `color` whose type is convertible to `Color`. The requirement that the data member needs to be mutable may be formulated by expecting the validity of an expression that assigns to it:

```
template<typename T>
virtual concept bool Shape = requires(T x)
{
    { x.color } -> Color;
    { x.color = Color{255, 0, 0} } -> Color&;
};
```

4.6. Associated types and dependent names

Concepts TS [6] allows specifying so-called *type requirements*, which mandate the validity of certain type names that can formed using the modeling type variable `T`. For instance, the concept definition below requires all types modeling the `Device` concept to have an associated type called `PartType`:

```
template<typename T>
concept bool Device = requires()
{
    typename T::PartType;
    auto T::add_part(PartType) -> void;
    // ...
};
```

A type models the `Device` requirement if it contains a `PartType` type alias in its definition, or a nested type named `PartType` (in either case, `PartType` must be publicly accessible). For example, the class `Robot` below satisfies that type constraint:

```
struct Joint
{
    Joint(std::string);
    auto get_name() const -> std::string;
    // ...
};

struct Robot
{
    using PartType = Joint;
    auto add_part(Joint) -> void;
    // ...
};
```

Using such concepts virtually is possible; however, generic algorithms using virtual concepts are compiled separately from the calling code in general, and do not see the definition of their arguments' types. For this reason, they cannot access type information in the same way as function templates do, and cannot navigate associated types *unless* those types are themselves abstracted by other concepts.

Hence, in order to allow for a meaningful virtual use of a given concept, type requirements should be accompanied by further constraints that specify what concept(s) an associated type models. The following `DevicePart` concept could be used as an abstraction of device parts:

```
template<typename T>
concept bool DevicePart = requires()
{
    T::T(std::string);
    auto T::get_name() const -> std::string;
    requires SemiRegular<T>;
    // ...
};
```

We could now add a constraint to our `Device` concept saying that the associated type `PartType` must be a model of `DevicePart`:

```
template<typename T>
concept bool Device = requires()
{
    typename T::PartType;
    requires DevicePart<typename T::PartType>;
    auto T::add_part(PartType) -> void;
    // ...
};
```

Generic algorithms using `Device` virtually can now create objects of its associated part type as instances of the `DevicePart` concept without knowing their concrete type:

```
void add_new_part(virtual Device& d, std::string name)
{
    Device::PartType p{std::move(name)};
    d.add_part(p); // THE COMPILER KNOWS THAT p HAS THE CORRECT TYPE
}
```

When no constraint is specified on the concept(s) modeled by an associated type, the associated type can only be used as a `Void` – see Section 4.2. In particular, this means that it must be used with reference semantics and it needs to be downcast to its exact type in order to perform any meaningful operation on it.

In general, there are limitations to the kind of types that can be formed in a concept which is meant to be used virtually, even when used as return type requirements. For instance, consider the following `Iterator` concept:

```
template<typename I>
concept bool Iterator = requires(I x)
{
    { *x } -> typename std::iterator_traits<I>::reference;
    { ++x } -> I&;
    requires CopyConstructible<F> && CopyAssignable<F> && Destructible<F>;
};
```

Here, the type name `std::iterator_traits<I>::reference` is dependent on the concrete type which models the concept, and similar considerations apply to those I have made above about associated types. Since the value of the type parameter `I` is unknown to a generic algorithm that uses `Iterator`, the correct instantiation of `std::iterator_traits<>` cannot be produced, and the return type of the expression `*x` must be treated as `Void` – unless the same type is subject to an additional constraint that requires it to model a certain concept, as in the case of `Device` and `DevicePart`.

4.7. Adaptation

In certain situations we need to treat polymorphically several objects whose types' definitions cannot be changed and do not follow syntactically consistent conventions (and client code may not know the type of its arguments at compile-time).

The decision of letting the compiler deduce the modeling relation between types and concepts based on syntactic match (also called *duck-typing*), despite being appropriate in most design situations [30], does not guarantee sufficient flexibility in the general case.

As an example, consider two libraries A and B, both of which include a type that models a person. `A::Person` and `B::Individual` expose a few semantically-equivalent member functions, but the names of those functions are different, and the values they return are also represented differently.

Class `A::Person` has two member functions named `get_first_name()` and `get_family_name()`, returning the corresponding components of a person's name:

```
namespace A
{
    class Person
    {
        Person(std::string first_name, std::string family_name);
        std::string get_first_name() const;
        std::string get_family_name() const;
        // ...
    };
}
```

Class `B::Individual` has one single function, `get_name()`, which returns a string that concatenates the three components of a person's name passed at construction time, separated by a white space:

```
namespace B
{
    class Individual
    {
        Individual(std::string first_name,
                   std::string middle_name,
                   std::string family_name);
        std::string get_name() const;
        // ...
    };
}
```

Given that we are not allowed to alter these two type definitions, how can we write a run-time generic algorithm that is capable of working with heterogeneous collections containing objects of those types, possibly mixing `A::Persons` with `B::Individuals`?

First of all we need to define the (virtual) concept in terms of which the generic algorithm will be written. A possible definition for such a concept may look as follows:

```
template<typename T>
virtual concept bool NamedPerson = requires()
{
    auto T::get_name() const -> std::string;
    auto T::get_surname() const -> std::string;
};
```

And this is how our generic algorithm could use it:

```
void print_all_data(std::vector<NamedPerson const*> const& people)
{
    for (auto p : people)
    {
        std::cout << "   Name: " << p->get_name() << std::endl;
        std::cout << "Surname: " << p->get_surname() << std::endl;
    }
}
```

The obvious problem is now how to adapt the interface defined by `NamedPerson` to those exposed by `A::Person` and `B::Individual`. This is where *adaptation* – which was known as *concept maps* in the context of C++0x [31] – comes into play. The adaptation logic necessary to make a given type model a given concept can be specified using the *syntax* (but not the semantics, as we shall see) of template specialization:

```
template<>
virtual concept bool NamedPerson<A::Person> = requires()
{
    auto get_name() const -> std::string
    { return this->get_first_name(); }

    auto get_surname() const -> std::string
    { return this->get_family_name(); }
};
```

Function definitions inside a concept adaptation are not restricted to bare delegations:

```
template<>
virtual concept bool NamedPerson<B::Individual> = requires()
{
    auto get_name() const -> std::string
    {
        auto whole_name = this->get_name();
        auto first_space = whole_name.find(' ');
        return whole_name.substr(0, first_space);
    }

    auto get_surname() const -> std::string
    { /* SIMILARLY... */ }
};
```

The `this` keyword used inside a function definition when specializing a concept refers to the actual object being erased. It can be used for disambiguating function calls in situations where both the concept and the modeling type have a function with the same name – omitting the `this` keyword would result in an invocation through the concept's interface, as discussed in Section 7.

Unlike in the primary definition of a concept, in a concept specialization no explicit qualification is needed for distinguishing member functions from freestanding functions: since freestanding functions can be added non-intrusively to the interface of a type, all functions defined in a concept specialization are implicitly assumed to implement member functions.

Similarly, when associated types must be non-intrusively added to the interface of an existing type, they can be defined (possibly as a type alias) directly inside a concept specialization:

```
template<typename T>
virtual concept bool NamedPerson = requires()
{
    typename T::AddressType;
    requires Printable<typename T::AddressType>;
    // ...
};

template<>
virtual concept bool NamedPerson<A::Person> = requires()
{
    using AddressType = std::string;
    // ...
};
```

We encountered an interesting use case for adaptation back in Section 4.4, when we defined the `Callable` concept like so:

```
template<typename F, typename Ret, typename... Args>
virtual concept bool Callable = requires()
{
    auto F::operator () (Args...) -> Ret;
    requires CopyConstructible<F> && Destructible<F>;
};
```

Back there I remarked that `Callable` is not modelled by `std::nullptr_t`, which makes it illegal to initialize a `Callable` with `nullptr`. As anticipated, this problem can be solved through adaptation:

```
template<typename Ret, typename... Args>
virtual concept Callable<std::nullptr_t, Ret, Args...>
{
    auto operator () (Args...) -> Ret
    {
        throw std::bad_function_call{};
    }
}
```

It is worth mentioning that not all functions required by a virtual concept *must* be explicitly adapted: if duck-typing is acceptable, and if a syntactic match exists for *some* functions between a particular type and the concept it is meant to model, only the non-matching ones need to be explicitly adapted.

4.8. Structural DGP and (the absence of) `virtual auto`

One of the new features introduced by Concepts TS [6] is the possibility of using the `auto` placeholder to write function templates with syntax similar to the one of regular functions – sometimes referred to as *terse notation*.

For instance, the following two definitions of `foo()` are meant to be equivalent, and they both declare a function *template*:

```
void foo(auto x) // TERSE NOTATION: UNCONSTRAINED FUNCTION TEMPLATE!
{
    x.bar();
    // ...
}

template<typename T>
void foo(T x)
{
    x.bar();
    // ...
}
```

When the `auto` type specifier appears as the type of a parameter in a function's parameter list, the generated function template is unconstrained on the corresponding parameter(s). In other words, an object will be accepted as the input of the generic algorithm `foo()` if and only if the usage of it made inside the body of `foo()` is valid: no early checking will be performed by the compiler based on the type of the argument before instantiating `foo()`.

This purely *structural* approach to GP has pros and cons: changing the definition of a function will not require consistently updating its interface; on the other hand, compiler diagnostics are likely to be obscure, and users of a function are forced to inspect its implementation in order to determine how it can be called, rather than being concerned with its interface only.

The kind of trade-offs between the structural and the *nominal* approach to GP (i.e. programming against explicit interfaces) is well analyzed in [30], where the importance of supporting both styles is stressed. In the context of DGP with virtual concepts, unfortunately, structural polymorphism makes little sense.

To see why, one should consider the fact that in order to infer a concept definition from the *usage* which is being made of a parameter inside a function and check whether the corresponding argument fulfills its requirements, the definition of that function must be visible at call site.

This, on the other hand, eliminates the possibility of separate compilation, making us pay the cost of the compile-time dependency *and* the cost of the run-time dispatch; in such a scenario, switching to static polymorphism with templates and constraints simply makes room for more optimization without losing on expressivity and flexibility. In a sense, the structural approach offered by `auto` and unconstrained generic algorithms is lost on virtual concepts.

One may object that this is not necessarily true in slightly more complex situations where the `auto` placeholder is used only as *part* of a parameter's type specifier. Consider:

```
void foo(std::vector<virtual auto*> const& v)
{
    for (auto elem : v)
    {
        elem->bar();
        // ...
    }
}
```

The programmer's intention here may be to allow invoking `foo()` with a *heterogeneous* vector of pointers (where the types of the pointed objects do not share a common interface), which would not be possible when using `auto` with regular function templates.

Unfortunately, this goal cannot be attained in practice, because clients would have to fill a `vector<C>` (for some given concept C) to provide as an argument to `foo()`, and such a vector would not be reference-compatible with a `vector<Unnamed>`, where `Unnamed` is the concept inferred from the body of `foo()`, [5] unless the two concepts were identical; but this would defeat the purpose of implicitly inferring that concept's definition, making `virtual auto` a technically valid, but practically useless feature.

4.9. Concept subsumption

A useful relation that can be defined between pairs of concepts is the relation of *subsumption*. In short, a concept `C1` subsumes another concept `C2` if and only if, given any type `T` which is a model of `C2`, `T` also happens to be a model of `C1`. Informally, the subsumption relation is to concepts what the "*is base class of*" relation is to class inheritance. In the following, I will write `C2` ≤ `C1` whenever concept `C1` subsumes concept `C2`. The dual relation is called *refinement*.

While the interplay between inheritance and concepts will be analyzed in deeper detail in Section 9, I will now briefly introduce a parallel in the way inheritance and virtual concepts support polymorphic manipulation of objects.

[5] The reason for this is similar to the reason why a `vector<D*>` cannot substitute a `vector<B*>` if D derives from B: adding a pointer to D2 to a vector of the latter type (where D2 derives from B, but not from D) is a sensible operation, but allowing the same on a `vector<D*>` would break the type system.

Consider the following two concepts:

```
template<typename T>
virtual concept bool Foo = requires()
{
    auto T::foo() -> void;
}

template<typename T>
virtual concept bool FooBar = requires()
{
    auto T::foo() -> void;
    auto T::bar() -> void;
}
```

Here, it is easy to see that `FooBar` ≤ `Foo`, because `FooBar` defines a strict superset of the requirements defined by Foo; hence, any type that satisfies `FooBar`'s requirements also satisfies `Foo`'s. Because of this, it is safe to use a `FooBar` as a `Foo` whenever doing so is either necessary or convenient. In particular, the following initialization is legal:

```
FooBar* fb = /* ... */;
Foo* f = fb; // OK!
```

Similarly, we ought to be able to pass an object of any type that models `FooBar` to any function that accepts a `Foo`:

```
void buzz(Foo f)
{
    f.foo();
    // ...
}

int main()
{
    FooBar* fb = /* ... */;
    buzz(*fb); // OK!
}
```

Notice that no slicing occurs here, even though we are passing the argument to `buzz()` by value: if necessary, the compiler will take care of setting up the necessary heap allocations to avoid it; – see Section 12 for a more detailed account.

As one can see, the relation of subsumption between concepts is non-intrusive and deduced by syntactic match: there is no need to modify a concept's definition to explicitly assert that it refines another concept. In fact, the compiler will be able to infer this when necessary by comparing the two concept definitions, matching one to the requirements of the other; all in all, this is the same procedure used to determine whether some concrete type models a given concept.

Consistently with the discussion in Section 4.8 as well as in Section 9, concept subsumption does not imply substitutability of templates that are parameterized by them. In other words, the following is illegal:

```
void foo(std::vector<Foo>& f)
{
    // ...
}

int main()
{
    std::vector<FooBar> v;

    // FILL IN v...

    foo(v); // ERROR! vector<Foo> AND vector<FooBar> ARE NOT RELATED
}
```

As explained in Section 12, erasing concepts through other concepts may result in a non-negligible performance overhead due to the layers of indirections necessarily introduced by the compilers. This is an inherent problem that cannot be avoided, and users should be aware of the performance trade-offs that it implies when designing their programs.

4.10. Polymorphic containers

Non-intrusive dynamic polymorphism opens up several new ways of designing algorithms. One interesting possibility is to have separately-compiled functions that can work with any container providing basic iteration functionalities.

For instance, consider the following (partial) concept definitions:

```
template<typename T, typename V>
virtual concept bool ForwardIterator = requires(T it)
{
    { *it } -> V&;
    { ++it } -> T&;
    // ...
};

template<typename T, typename V>
virtual concept bool SequentialContainer = requires(T c)
{
    { c.begin(); } -> ForwardIterator<V>;
    { c.end(); } -> ForwardIterator<V>;
    // ...
}
```

It is clear that, for some given T, the `SequentialContainer<T>` concept (or at least its visible part from the snippets above) is modeled both by `std::vector<T>` and by `std::list<T>`. The `ForwardIterator<T>` concept erases the iterators of those containers, which allows writing generic algorithms like `print_all()` below:

```
void print_all(SequentialContainer<int> const& c)
{
    for (auto x : c)
    {
        std::cout << x << " ";
    }
}
```

This algorithm can be compiled separately from the rest of the code, and can be invoked like so:

```
int main()
{
    std::vector<int> v{1, 2, 3};
    print_all(v);

    std::list<int> l{4, 5, 6};
    print_all(l);
}
```

4.11. Run-time generic algorithms

The example from Section 4.10 shows how to erase a container's type and write a generic algorithm that only knows about the type of the contained values; however, erasing the type of the container is not enough if we want to write truly generic algorithms – i.e. akin to the algorithms from the Standard Library.

Let's consider the following two concepts (`Callable` is taken from Section 4.4):

```
template<typename T>
virtual concept bool InputIterator = requires(T it)
{
    { ++it } -> T&;
    { *it } -> ValueType&;
    typename InputIterator::ValueType;
    // ...
};

template<typename F, typename Ret, typename... Args>
virtual concept bool Callable = requires()
{
    auto F::operator () (Args...) -> Ret;
    requires CopyConstructible<F> && Destructible<F>;
};
```

Here, the `InputIterator` concept is not parameterized explicitly by the concrete value type (unlike the `ForwardIterator<T>` concept from Section 4.10). Because of this, as explained in Section 4.6, the associated type `ValueType` can only be treated as a `Void` by generic algorithms.

I have shown in Section 4.2 that generally, objects that are accessed through a `Void` concept cannot be manipulated in any meaningful way other than downcasting them to their concrete type. While this is true in general, there are specific situations where the compiler has just enough information to deduce that the underlying object has a type with certain properties (e.g. it is the same type as the type of another erased object), and therefore can perform meaningful actions with it.

For example, here is how we could write a non-template, dynamic version of the `std::for_each()` algorithm:

```
auto for_each(InputIterator first,
              InputIterator last,
              Callable<void, typename InputIterator::ValueType const&> F)
{
    for (; first != last; ++first)
    {
        f(*first); // TYPE-SAFE, YET RESOLVED AT RUN-TIME!
    }

    return f;
}
```

Even though the body of `for_each()` does not know anything at all about the value type of the iterator, it knows that F's call operator takes an object of *that* type by reference: therefore, the call is verified for type-safety at compile time, but dispatched at run-time.

If we wanted to get fancy, we could even use adaptation to let the `int` type model the `InputIterator` concept:

```
template<>
concept bool InputIterator<int> = requires()
{
    auto operator ++ () -> int& { ++(*this); return *this; };
    auto operator * () -> Value& { return *this; }
    using Value = int;
    // ...
};
```

This would allow us to invoke `for_each()` like so:

```
for_each(0, 10, [] (int i) { std::cout << i; });
```

While the usefulness and elegance of the above example are dubious, this example does demonstrate the potential of virtual concepts for DGP in C++.

4.12. Solving the Expression Problem in C++

I shall conclude this Section by showing how virtual concepts can be used to solve the Expression Problem [1] in C++. The Expression Problem is an engineering puzzle first formalized by Philip Wadler:

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

In traditional OOP, a design that allows adding new data types without requiring recompilation of generic algorithms is achieved by *(a)* creating an abstract class (interface) `I` with the necessary operations, *(b)* letting concrete types derive from `I` and provide an implementation for those abstract operations, and *(c)* letting generic algorithms work with (pointers or references to) instances of `I`.

However, in such a setting, adding new *functions* capable of working polymorphically with the existing types requires modifying the definition of `I` as well as the definitions of its sub-types; incidentally, this also means recompiling all the generic algorithms that depend on `I`'s definition.

Alternative OOP techniques, the most popular being the Visitor design pattern [32], allow adding new polymorphic functions capable of working with existing types without requiring any modification to those types' definitions. However, adding new *types* for polymorphic usage by those functions will require recompiling everything.

All in all, the programmer is forced to make a choice between extensibility along the dimension of functions and extensibility along the dimension of types. How to achieve both is the essence of the Expression Problem.

Unfortunately, C++ does not currently offer language constructs for solving this problem. This does not mean that the problem *cannot* be solved: experience from other languages shows that offering this support is feasible and useful. For instance, *type classes* in Haskell [33] allow extending the set of interfaces a type adheres to without modifying its definition – an interesting discussion of how to solve the Expression Problem in Haskell is given in [34]. Type classes mostly differ from virtual concepts in that they always require an explicit mapping of types to the concepts they model.

Rust's traits [35] provide similar functionality, and also let the programmer choose on a case-by-case basis whether they should be used for static (compile-time) or dynamic (run-time) polymorphism – a design choice I adopted for virtual concepts too, as discussed in Section 3.5.

Similar to type classes in Haskell and traits in Rust, virtual concepts realize non-intrusive dynamic polymorphism in C++; this, in turn, allows solving the Expression Problem without resorting to complicated library solutions such as [9]. To show how one could go about it, I shall now introduce a working example.

The following types are meant to represent the nodes of an expression tree for some simple arithmetic language featuring constant values, additions, and subtractions:

```
struct value
{
    int val;
};

template<typename L, typename R>
struct plus
{
    L lhs;
    R rhs;
};

template<typename L, typename R>
struct minus
{
    L lhs;
    R rhs;
};
```

Let's also suppose we have a set of freestanding function overloads that, given an expression, allow evaluating its result, so that `value`, `plus`, and `minus` model the `expression` concept:

```
auto evaluate(value const& v)
{
    return v.val;
}

template<typename L, typename R>
auto evaluate(plus<L, R> const& e)
{
    return evaluate(e.lhs) + evaluate(e.rhs);
}

template<typename L, typename R>
auto evaluate(minus<L, R> const& e)
{
    return evaluate(e.lhs) - evaluate(e.rhs);
}
```

In a separate translation unit we can then define (and make use of) the `Evaluatable` concept:

```
template<typename T>
concept bool Evaluatable = requires()
{
    auto evaluate(T const&) -> double;
};

void print_result(virtual Evaluatable const& e)
{
    std::cout << "Result = " << evaluate(e) << std::cout;
}
```

With proper overloads of the arithmetic addition and subtraction operators to provide convenient syntax for creating instances of `plus` and `minus`, we could write programs such as:

```
int main()
{
    auto expr = value{1729} + (value{1337} - value{42});

    print_result(expr); // PRINTS "Result = 3024"
}
```

In order to show how virtual concepts allow solving the Expression Problem, we shall first extend the set of types that can be handled polymorphically by `print_result()` without recompiling any of the existing code; then, we will add a new polymorphic function that can work with the existing expression types, again without requiring existing code to be recompiled.

Introducing a new pair of expression templates `multiply` and `divide` is trivial:

```
template<typename L, typename R>
struct multiply
{
    L lhs;
    R rhs;
};

template<typename L, typename R>
struct divide
{
    L lhs;
    R rhs;
};

template<typename L, typename R>
auto evaluate(multiply<L, R> const& e)
{
    return evaluate(e.lhs) * evaluate(e.rhs);
}

template<typename L, typename R>
auto evaluate(divide<L, R> const& e)
{
    return evaluate(e.lhs) / evaluate(e.rhs);
}
```

Without recompiling `print_result()`, we can now invoke it with a more complex input:

```
int main()
{
    auto expr = value{3} * value{4} + (value{5} - value{1}) / value{2};

    print_result(expr); // PRINTS "Result = 14"
}
```

Once again, we have been assuming the existence of overloaded multiplication and division operators for syntactic convenience – their definition is trivial but not shown for brevity.

What we did so far is unlikely to impress anyone who is used to inheritance-based polymorphism: the ability to support new polymorphic types without recompiling generic functions has always been the strong point of inheritance. However, the flexibility of virtual concepts shines when extending the set of *functions* that can work polymorphically with our existing types, again without recompiling existing code. Consider:

```
template<typename T>
concept bool StreamInsertable = requires(T expr, std::ostream out)
{
    { out << expr } -> std::ostream&;
};

auto operator << (std::ostream& out, value const& v) -> std::ostream&
{
    return out << v.val;
}

template<typename L, typename R>
auto operator << (std::ostream& out, plus<L, R> const& e) -> std::ostream&
{
    return out << "(" << expr.lhs << " + " expr.rhs << ")";
}

template<typename L, typename R>
auto operator << (std::ostream& out, minus<L, R> const& e) -> std::ostream&
{
    return out << "(" << expr.lhs << " - " expr.rhs << ")";
}

template<typename L, typename R>
auto operator << (std::ostream& out, multiply<L, R> const& e) -> std::ostream&
{
    return out << "(" << expr.lhs << " * " expr.rhs << ")";
}

template<typename L, typename R>
auto operator << (std::ostream& out, divide<L, R> const& e) -> std::ostream&
{
    return out << "(" << expr.lhs << " / " expr.rhs << ")";
}
```

With all of this in place, we can now write new run-time generic functions like `print_expression()` below, whose definition can be put in a separately-compiled translation unit:

```
void print_expression(virtual StreamInsertable const& e)
{
    std::cout << "Expression = " << e << std::cout;
}
```

Finally, we can invoke both `print_expression()` and `print_result()` from `main()` like so:

```
int main()
{
    auto expr = value{3} * value{4} + (value{5} - value{1}) / value{2};

    print_expression(expr); // PRINTS "Expression = (3 * 4) + ((5 – 1) / 2)"

    print_result(expr); // PRINTS "Result = 14"
}
```

In summary, we managed to extend our design both in the dimension of types and in the dimension of functions without requiring any modification to the existing code, which is what the Expression Problem is all about.

5. Concept interface specification

The main purpose of this Section is to illustrate the algorithm that allows, given a concept definition, to extract what I call its *erasing interface*.

The erasing interface defines what clients dealing with an object erased by a given virtual concept are allowed to do with that object. In particular, the erasing interface determines the set of functions that can be invoked on that object, their return types, the types of their parameters, and (obviously) their names.

A concept's erasing interface is also used to decide whether a given type satisfies the requirements of that concept: the details of this decision procedure, which is based on the idea of *forwarding thunks*, are also presented in this Section.

5.1. Erasing interfaces

A concept specifies one or more requirements that modeling types must satisfy. Every concept used virtually has an associated *erasing interface* which is derived from those requirements. Given a concept definition, the erasing interface is determined as explained below.[6]

As a preliminary step, I define the *forwarding type* of an expression `e` as the result of invoking the macro `ERASING_TYPE(e)`, whose definition is given below:

```
namespace detail
{

    template<typename T>
    struct holder
    {
        using type = T;
    };

    template<typename T>
    auto erasing_type(T&& x)
    {
        return holder<T&&>{};
    }

}

#define ERASING_TYPE(x) decltype(detail::erasing_type(x))::type
```

[6] Note that an erasing interface may have more functions than the number of requirements in a virtual concept definition.

In short, for each *lvalue* expression of type A, the corresponding forwarding type is A&, whereas for each *rvalue* expression of type A, the corresponding forwarding type is A&&.

Forwarding types are used to determine the parameter types of the function associated to a given compound requirement in a given concept definition. Because of how I defined forwarding types above, the signatures of functions in the erasing interface of a concept will always have parameters of reference type.

For instance, none of the following static assertions should fire:

```
int x = 0;

int&& y = 42;

static_assert(std::is_same<ERASING_TYPE(x), int&>{});

static_assert(std::is_same<ERASING_TYPE(y), int&>{});

static_assert(std::is_same<ERASING_TYPE(std::move(x)), int&&>{});

static_assert(std::is_same<ERASING_TYPE(std::move(y)), int&&>{});

static_assert(std::is_same<ERASING_TYPE(42, int&&>{});

static_assert(std::is_same<ERASING_TYPE(int{}, int&&>{});
```

Given a compound requirement R whose expression contains an invocation of a free function F with arguments `e1`, `e2`, …, `eN` and whose constraint on the expression's return type is T, the *erasing signature* corresponding to R is given by invoking the macro `ERASING_SIGNATURE(T, e1, e2, …, eN)`, defined like so:

```
namespace detail
{

    template<typename Ret, typename... Ts>
    auto erasing_signature(Ts&&... args)
    {
        return holder<
            Ret(typename ERASING_TYPE(std::forward<Ts>(args))...)
            >{};
    }

}

#define ERASING_SIGNATURE(Ret, ...) \
    decltype(detail::erasing_signature<Ret>(__VA_ARGS__))::type
```

In simpler terms, this amounts to defining the erasing signature for requirement R as:

```
T(ERASING_TYPE(e1), ERASING_TYPE(e2), …, ERASING_TYPE(en))
```

Given the definitions above, none of the following static assertions shall fire:

```
int x = 0;

int&& y = 42;

static_assert(std::is_same<ERASING_SIGNATURE(bool, y), bool(int&)>{});

static_assert(std::is_same<ERASING_SIGNATURE(bool, 42, y),
                           bool(int&&, int&)>{});

static_assert(std::is_same<ERASING_SIGNATURE(bool, x, y),
                           bool(int&, int&)>{};

static_assert(std::is_same<ERASING_SIGNATURE(bool, std::move(x), y),
                           bool(int&&, int&)>{});

static_assert(std::is_same<ERASING_SIGNATURE(bool, int{}, y),
                           bool(int&&, int&)>{});
```

Requirements that contain a member function call expression (either through the arrow operator or the dot operator) are handled similarly, with the expression yielding `*this` being used as an additional first argument. For example, considering the following concept definition:

```
template<typename T>
concept bool Simple = requires(T x, int y)
{
    { foo(x, y) } -> bool;
    { x.bar(y) } -> void;
};
```

The erasing interface for the `Simple` concept will define two erasing functions, like so:

```
[f] bool foo(T&, int&)
[m] void bar(T&, int&)
```

If the expression of type T yielding `*this` is constant, then the first argument of the erasing function will be a reference to `const T`. Consider this slightly modified version of the `Simple` concept:

```
template<typename T>
concept bool Simple2 = requires(T x, T const y, int z)
{
    { foo(x, z) } -> bool;
    { y.bar(z) } -> void;
};
```

The corresponding erasing interface will include the following entries:

```
[f] bool foo(T&, int&)
[m] void bar(T const&, int&)
```

The annotation preceding each entry is used to keep track of the nature of the function – i.e. whether the erased function is a free function or a member function.

5.2. Forwarding thunks

Given a function `f` in the erasing interface of some given concept `C` and a type `T`, we can define a *forwarding thunk for* `f` *towards* `T` – and denote it as `thunk(f, T)` – as follows.

The forwarding wrapper has the same signature as `f` and its body consists of a single return statement. I shall denote the parameters of `f` as `p1, …, pN`. If `f` is a free function, the returned expression is a function call `f(FWD(p1), …, FWD(pN))`; if `f` is a member function, the returned expression is `p1.f(FWD(p2), …, FWD(pN))`.[7] In both cases, for any parameter `p`, the `FWD()` macro is defined like so:

```
#define FWD(x) std::forward<decltype(x)>(x)
```

For example, consider the erasing interface for the `Simple2` concept from Section 5.1 and type `fizz` below:

```
struct fizz
{
    auto bar(int& x) { x = 42; }
    auto buzz() const { return 1337; }
};

auto foo(fizz const& f, int const i)
{
    return (i > f.buzz());
};
```

This is how the forwarding thunks for functions `foo()` and `bar()` towards `fizz` would be defined according to the procedure described above:

```
auto foo_thunk(fizz const& p1, int& p2)
{
    return foo(std::forward<decltype(p1)>(p1),
               std::forward<decltype(p2)>(p2));
}

auto bar_thunk(Fizz& p1, int& p2)
{
    return std::forward<decltype(p1)>(p1).bar(std::forward<decltype(p2)>(p2));
}
```

This is basically the same forwarding logic used by `std::function` and all `INVOKE`-based facilities to pass their arguments to the wrapped callable objects.

[7] Clearly, if N is less than two, the function call is performed with no arguments.

5.3. The modeling relation

Given a type T and a concept C whose erasing interface contains entries for functions F1, F2, …, Fn, T is said to be a *model* of C if and only if, for each function `Fi`, `thunk(Fi, T)` is well-formed.

With reference to the example from Section 5.2, `fizz` is a model of `Simple2`, while the following type `fuzz` is not, because the member function `bar()` takes its argument by *rvalue reference*, while `thunk(bar)` would forward its own corresponding parameter as an *lvalue*:

```
struct fuzz
{
    auto bar(int&& x) { x = 42; } // PROBLEM HERE!
    auto buzz() const { return 1337; }
};

auto foo(fizz const& f, int const i)
{
    return (i > f.buzz());
};
```

One important thing to keep in mind is that the signatures of functions in the erasing interface of a concept are *not* necessarily the signatures that modeling types have to support in order to satisfy that concept. Consider the erasing signature below:

```
[m] void bar(T const&, int&&)
```

This requires the presence of a `const`-qualified member function called `bar()` in the modeling type. However, that function does not need to take an *rvalue reference* of type `int`. The parameter type can be `int`, `int&&`, or anything that can be copy-initialized from an rvalue or type `int`. Signatures 1-5 below, for instance, all satisfy the requirement, while signatures 6-7 do not:

```
struct fizz
{
    fizz(int) { }
};

struct buzz
{
    explicit buzz(int) { }
};

void bar(int&&) const   // #1 - OK
void bar(int) const     // #2 - OK
void bar(double) const  // #3 - OK
void bar(char&&) const  // #4 - OK
void bar(fizz) const    // #5 - OK

void bar(int&) const    // #6 - ERROR
void bar(buzz) const    // #7 - ERROR
```

If a type T models a concept C per the definition given above and an attempt is made to erase an object of type T through a variable of type C (or const-qualified pointer or reference to C)[8], the compiler will generate a corresponding *instantiation table* containing the addresses of all the forwarding thunks generated for each function in the C's erasing interface towards T. In the following I will refer to the instantiation table of a type T for concept C as `itable(T, C)`.

An instantiation table can be thought of as a vtable. However, while vtable pointers are typically part of an object's representation, itable pointers are kept separate from the erased object. Moreover, as I will show in Section 6.2, instantiation tables can contain entries for constructors and static functions, which cannot be virtual and therefore are not referenced by traditional vtables.

For example, consider the definition of concept `Simple2` from Section 5.1 and the corresponding erasing interface (reported below for ease of reference):

```
template<typename T>
concept bool Simple2 = requires(T x, T const y, int z)
{
    { foo(x, z) } -> bool;
    { y.bar(z) } -> void;
};
```

As we saw, this concept definition has the following associated erasing interface:

```
[f] bool foo(T&, int&)
[m] void bar(T const&, int&)
```

In Section 5.2 we then introduced the `fizz` type (modeling `Simple2`) and its forwarding thunks:

```
struct fizz
{
    auto bar(int& x) { x = 42; }
    auto buzz() const { return 1337; }
};

auto foo(fizz const& f, int const i)
{
    return (i > f.buzz());
};

auto foo_thunk(fizz const& p1, int& p2)
{
    return foo(std::forward<decltype(p1)>(p1),
               std::forward<decltype(p2)>(p2));
}

auto bar_thunk(Fizz& p1, int& p2)
{
    return std::forward<decltype(p1)>(p1).bar(std::forward<decltype(i)>(i));
}
```

[8] Sections 6.1 and 6.2 explain these usage patterns in detail.

Figure 4 below provides a schematic view of the instantiation table of `fizz` for `Simple2`. The instantiation table contains two entries – one for each function in the erasing signature; each entry holds the address of the forwarding thunk generated by the compiler for the corresponding function.

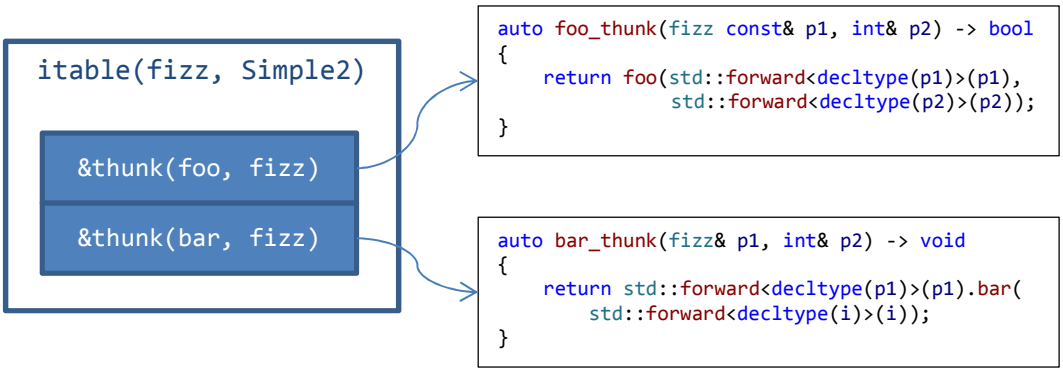

Figure 4 - Schema of a simple instantiation table

When only compound requirements are used in a concept definition – as was the case back in Section 5.1 – the instantiation table will contain as many entries as the number of requirements in the concept definition. This, however, no longer holds in general when signature requirements (see Section 4.3) are used, as explained in the next Section.

5.4. Signature requirements

Compound requirements based on expressions may be convenient in certain situations, but present problems in others. As briefly anticipated in Section 4.3, expression requirements for a given function call cannot specify, within a single expression, that both *lvalues* and *rvalues* are acceptable for any of its arguments; this is because expressions always fall in exactly one of these two categories.

When we want to express that the function which is the subject of the requirement can be invoked with *lvalue* and *rvalue* expressions for N of its arguments, we need 2^N compound requirements. This combinatorial explosion is an issue for maintenance. Unlike compound requirements, signature requirements allow *directly* specifying the signature of forwarding thunks; the syntax used for doing so is easy to extend in a way that combinations of value categories in a single requirement are supported. Consider, once again, the definition of `Simple2` from Section 5.1:

```
template<typename T>
concept bool Simple2 = requires(T x, T const y, int z)
{
    { foo(x, z) } -> bool;
    { y.bar(z) } -> void;
};
```

The equivalent definition using signature requirements would look as follows:

```
template<typename T>
concept bool Simple2 = requires()
{
    auto foo(T&, int&) -> bool;
    auto T::bar(int&) & -> void;
};
```

The requirement for `foo()` does not carry particular surprises: it reflects the signature of the corresponding forwarding thunk discussed in Section 5.2. Things are a bit more complicated for the requirement involving `bar()`. Firstly, the member function syntax needs to be used in order to specify that `bar()` is supposed to be a member function. Secondly, since the original requirement invokes the dot operator on an lvalue expression (`y`), the `&` ref-qualifier must be added to the signature requirement.

The latter part ensures that a type with a function `bar()` that can be called *only* on lvalue expressions can still model `Simple2` – that's what the original definition of that concept expressed. However, such a requirement is often inappropriate: in most cases, we want to specify that it should be possible to invoke member functions both on lvalues and on rvalues. The way to go about this when using expression requirements is to define *two* requirements for each such member function, as shown below:

```
template<typename T>
concept bool Simple2 = requires(T x, T const y, int z)
{
    { foo(x, z) } -> bool;
    { y.bar(z) } -> void;
    { std::move(y).bar(z) } -> void;
};
```

Things get worse when we also want to allow function arguments to be lvalue expression as well as rvalue epxressions; if that were the case for the definition above, we would need *four* compound requirements:

```
template<typename T>
concept bool Simple2 = requires(T x, T const y, int z)
{
    { foo(x, z) } -> bool;
    { y.bar(z) } -> void;
    { y.bar(42) } -> void;
    { std::move(y).bar(z) } -> void;
    { std::move(y).bar(42) } -> void;
};
```

It is not difficult to see where this is going: a member function with three parameters would lead to 16 compound requirements.

Signature requirements allow getting rid of this problem in two ways: the first possibility consists in *(a)* specifying that the arguments shall be taken by value and *(b)* omitting ref-qualifiers. This requires the arguments' types to be movable, but if that's acceptable the simplicity of this approach might make it the preferred choice:[9]

```
template<typename T>
concept bool Simple3 = requires()
{
    auto foo(T&, int&) -> bool;
    auto T::bar(int) -> void;
};
```

Consider type `fizz` from Section 5.2, whose definition is reported below for ease of reference:

```
struct fizz
{
    auto bar(int& x) { x = 42; }
    auto buzz() const { return 1337; }
};

auto foo(fizz const& f, int const i)
{
    return (i > f.buzz());
};
```

This type is *not* a model of `Simple3` with respect to the definition given above, because `thunk(bar, fizz)` (see Section 5.2) would be attempting to *move* the integer argument while passing it to `fizz::bar()`, which on the other hand takes its argument by mutable lvalue reference:

```
auto bar_thunk(Fizz& p1, int p2)
//                       ^^^ Note: argument taken by value
{
    return std::forward<decltype(p1)>(p1).bar(std::forward<decltype(p2)>(p2));
    //                                        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    //                                        Can't bind rvalue expression
    //                                        to mutable lvalue reference!
}
```

To re-establish the modeling relation, we are now forced to let `fizz::bar()` take its argument by value, which is what we desired:[10]

```
struct fizz
{
    auto bar(int x) { x = 42; }             // Now fizz is a model of Simple3
    auto buzz() const { return 1337; }
};
```

[9] I will have a lot more to say on movability of erased types and value semantics in general in Section 6.2.

[10] Because of how forwarding works, `fizz` would still model `Simple3` if it `bar()` took its argument by *rvalue* reference. What matters, however, is that clients are able to provide both lvalue and rvalue expressions as arguments, because of the copying which implicitly happens when passing the argument to the forwarding thunk.

The absence of ref-qualifiers in the signature requirement is tricky though: the above thunk definition takes its argument by (mutable) lvalue reference, but our intention was to be able to invoke `bar()` both on lvalue expressions *and on rvalue expressions*. To support this, signature requirements that do not specify any ref-qualifier actually generate *two* forwarding thunks with identical definition, but different signature:[11]

```
auto bar_thunk_l(Fizz& p1, int p2)
{
    return std::forward<decltype(p1)>(p1).bar(std::forward<decltype(p2)>(p2));
}

auto bar_thunk_r(Fizz&& p1, int p2)
{
    return std::forward<decltype(p1)>(p1).bar(std::forward<decltype(p2)>(p2));
}
```

This means the erasing interface for `Simple3` also contains *two* entries for `bar()`, and so do all instantiation tables for `Simple3`. Figure 5 below provides a schematic representation of `itable(fizz, Simple3)`:

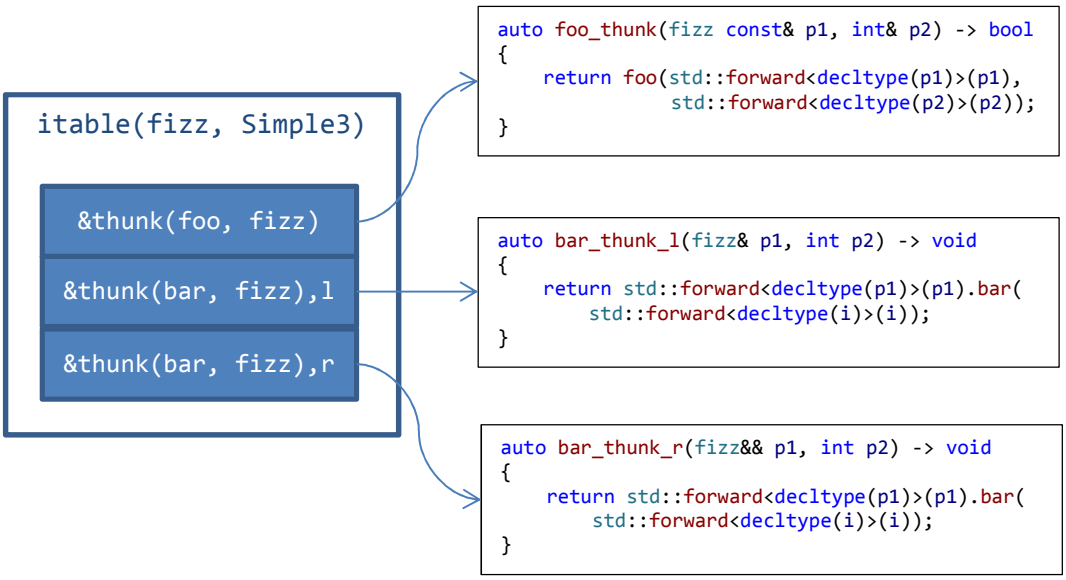

Figure 5 - Instantiation table of type `fizz` for concept `Simple3`.

Taking arguments by value in signature requirements has two possible downsides: (1) it requires those arguments' types to be movable and (2) it might have consequences in terms of performance – some types cannot be moved efficiently, and paying for an extra copy might not be acceptable in certain applications.

[11] The `_l` and `_r` suffixes – which, quite intuitively, stand for *rvalue* and *lvalue* – are arbitrary. Forwarding thunks are compiler-generated and their names are irrelevant to users. What matters here is to acknowledge the existence of several different forwarding thunks that are all generated for the same signature requirement.

As already mentioned in Section 4.3, the only alternative to taking arguments by value (provided that we want to avoid the combinatorial explosion of requirements) is to use the `&&&` modifier for those parameters that could be bind to both lvalues and rvalues. This causes the generation of two forwarding thunks – similarly to what happens for the `*this` expression as shown above – for each parameter decorated with the modifier. Consider the following concept definition:

```
template<typename T>
concept bool Simple4 = requires()
{
    auto foo(T&&&, int) -> bool;
    auto T::bar(int&&&) -> void;
};
```

The erasing interface for `Simple4` will contain six entries: two of those entries correspond to the requirement for `foo()`, while the remaining four correspond to the requirement for `bar()`:

```
[f] bool foo(T&, int)
[f] bool foo(T&&, int)
[m] void bar(T&, int&)
[m] void bar(T&&, int&)
[m] void bar(T&, int&&)
[m] void bar(T&&, int&&)
```

Consider now the definition of `fizz` given below:

```
struct fizz
{
    auto bar(int x) { x = 42; }
    auto buzz() const { return 1337; }
};

auto foo(fizz const& f, int const i)
{
    return (i > f.buzz());
};
```

Here, `fizz` is a model of `Simple4`, because all of the forwarding thunks generated for the erasing signatures reported above are legal. Figure 6 shows how `fizz`'s instantiation table for `Simple4` would look like.

One may wonder why this combinatorial explosion of automatically-generated overloads is indeed necessary: after all, C++ has function templates and forwarding references. The reason here is similar to the one which prevents function templates from being virtual: instantiation tables need to contain the address of each forwarding thunk for the functions in the erasing signature, and clients need to be able to invoke these functions by simply following this pointer; the number and type of arguments need to be agreed upon in advance, together with the position of each entry in the instantiation table.

Templates do not permit this. Taking the address of a function *template* is not possible, because different instantiations of it may give rise to completely different code: which of the infinitely many possible instantiations should be referenced when taking the address of a function template?

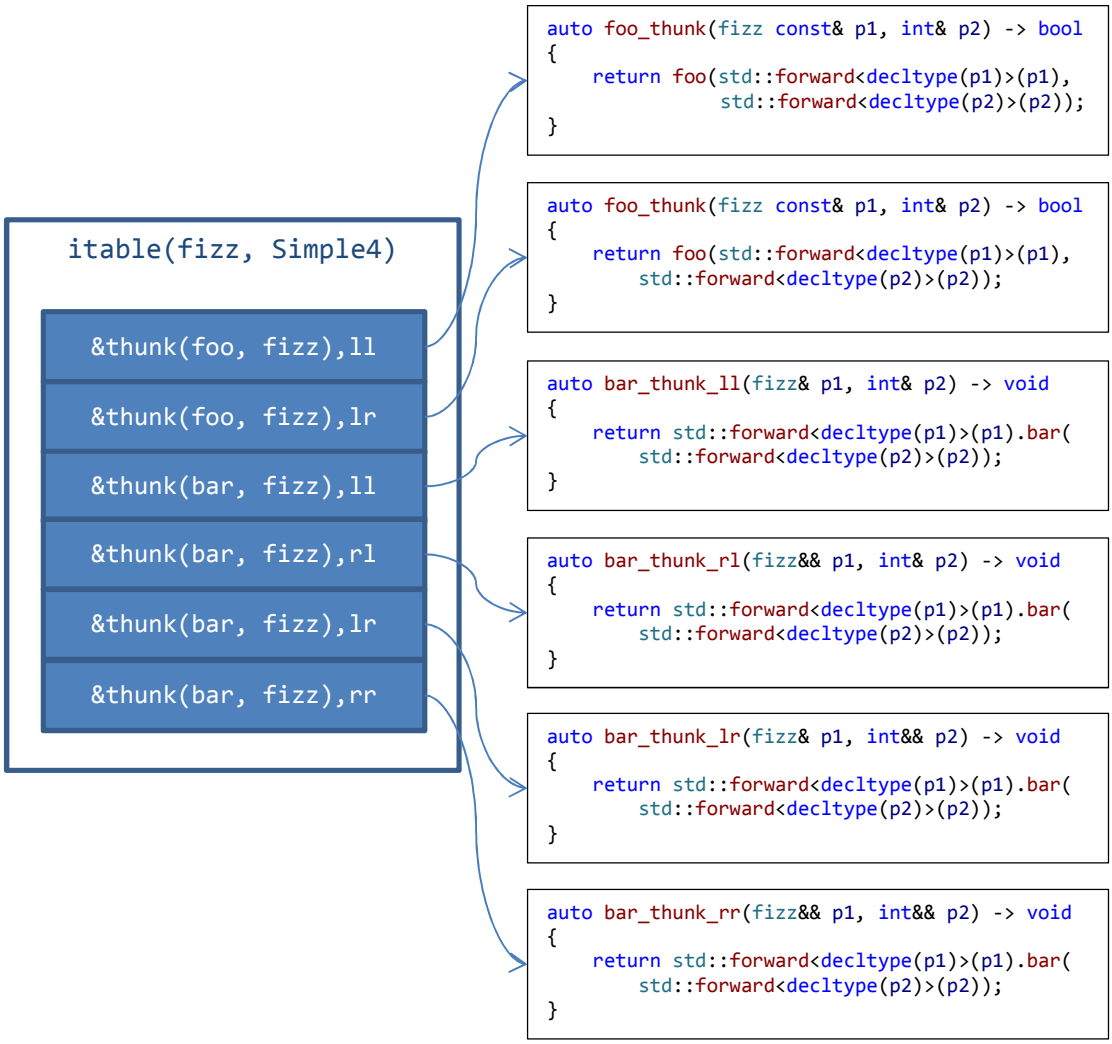

Figure 6 - Instantiation table of type `fizz` for concept `Simple4`.

With overloads auto-generated this way, the client code and the forwarding thunks can be compiled independently. What is important is that both sides are capable of agreeing on the signature of each forwarding thunk and on their index within instantiation tables for any given concept. This only requires that both parties see the definition of that concept.

Notably, the decision of which function to call (through the instantiation table) given certain arguments happens entirely at compile-time, without any run-time search overhead. More on this will be said in Section 6.

5.5. Special member functions

As stated in Section 3.2, one of the fundamental design goals for this proposal is to support both reference semantics and value semantics. Both will be discussed in greater detail in Section 6; however, it is worth expanding at this point on one implication of supporting value semantics that affects erasing interfaces and instantiation tables.

Supporting the virtual usage of concepts with value semantics means that code like the following shall be valid and behave correctly:

```
void foo(virtual Shape s)
{
    Shape r = s;                              // #1
    r.scale(42.0);                            // #2
    s = r;                                    // #3
    assert(s.get_area() == r.get_area());     // #4
}
```

Here, the object erased by `r` will have to be an instance of the same type of the object erased by `s`, copy-constructed from it (line 1). The occurrence of the `Shape` concept name, in fact, implies that the object erased by `r` is of the same type as the type of the object erased by parameter `s` (which is also declared as `Shape`). This is consistent with what we wrote in Section 4.1, as well as with the design decisions in Concepts TS [6]: repeated occurrences of the same concept name denote identical erased types. If this were not the case, it would be impossible to tell which of the potentially many models of `Shape` should be instantiated (and initialized from `s`) on line 1. Hence, the `Shape` concept shall only require copy-constructability.

Lines 2 and 4 are simple: the former requires the presence of a `scale()` member function taking an argument of a type `double` can be implicitly converted to; the latter requires the presence of a `get_area()` function returning anything which is equality-comparable. If the type identifiers used when declaring `r` and `s` were not identical, this line would require each type modeling `Shape` to be assignable to *any* other type modeling `Shape`. That is clearly not what we want, since a `Rectangle` is not meaningfully assignable to a `Circle`, a `Triangle` is not assignable to a `Rectangle` – and so on. However, because of what we wrote in the previous paragraph, the types of the objects erased by `r` and `s` must be identical, and the compiler knows this. Hence, all that is required here is that each type modeling `Shape` is assignable *to itself* (i.e. copy-assignable):

```
template<typename T>
virtual concept bool Shape = requires(T const& x, T y)
{
    T{}; // Default-constructible and destructible
    T{x}; // Copy-constructible (and destructible)
    y = x; // Copy-assignable
    { x.get_area() } -> double;
    { x.scale(2.0) } -> void;
};
```

The last remark worth making about the snippet above is that there is a hidden requirement: the type erased by `Shape` must be destructible.[12] Although no explicit usage of the destructor is made, when the `Shape` objects with automatic storage duration go out of scope, the corresponding destructor might be invoked. This is in line with the design goals stated in Sections 3.2 and 3.5.

But how does the compiler find the destructor of the erased object? And how can it find a copy-assignment operator, a copy-constructor or – even trickier – a default constructor? The answer is that all these functions (as well as other special member functions) can be part of the erasing interface of a concept, and itables can contain entries for them.

Since the size of an erased `Shape` object is unknown to our `foo()` function, construction of new objects must be done through dynamic allocation (modulo SBO, as discussed in Section 12). Consider, for instance, the following `Circle` class:

```
struct Circle
{
    Circle();
    Circle(Point center, double radius);
    Circle(Circle const& rhs);
    Circle& operator = (Circle const& rhs);
    double get_area() const;
    void scale(double f) const;
private:
    // ...
};
```

`Circle` is clearly an instance of `Shape`, and its itable is depicted in Figure 7 below. Constructor thunks perform dynamic allocations, while destructor thunks perform dynamic deletions.

It is important to stress that default-construction of a `Shape` object does not create an "empty" erasure wrapper such as `std::function`. Instead, it allocates a new object of the erased type and – therefore – requires models of `Shape` to be default-constructible themselves.

A related question is how the `foo()` function gets access to `Circle`'s itable for `Shape`. With regular, inheritance-based dynamic polymorphism, the dependency inversion is allowed by the vtable pointer, which is part of the object. A default constructor, on the other hand, is supposed to create a new object out of the blue, and there might be no existing object that would carry a vtable pointer to dereference. How does the dispatch happen?

One simple solution[13] is to have functions which are parameterized by some virtual concepts take one itable pointer as a hidden parameter for each such concept. On the calling side, the compiler is responsible for providing the correct itables based on the types of the arguments.

[12] This is arguably a requirement that all types should satisfy, and which is expressed by the first constraint in our last definition of `Shape`.
[13] See Section 12 for a more detailed account.

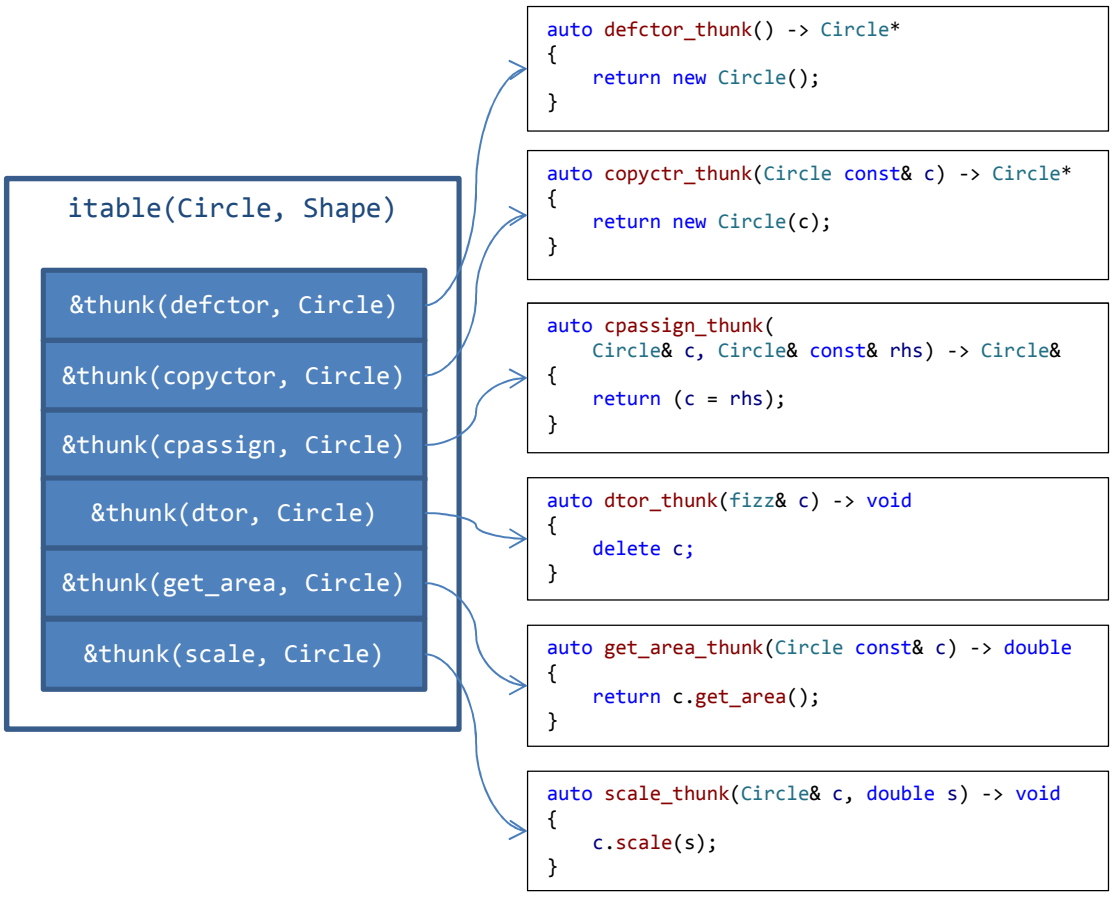

Figure 7 – Instantiation table of class `Circle` for concept Shape.

When a function call involving a concept-typed object is found, the compiler will find the corresponding itable among the implicit arguments and look for the appropriate entry; the thunk is then invoked by an indirect call through that pointer.

Instantiation table entries for move-constructors and move-assignment operators are treated similar to copy-constructors and copy-assignment operators, with thunks duly modified according to the rules presented in Section 5.2.

A tricky aspect to take into account when generating forwarding thunks for constructors is the distinction between brace-initialization and regular initialization. The two syntaxes can have different outcomes based on the constructors offered by a type; given the same arguments, it is also possible for one to be well-formed while the other is not.[14] Therefore, a forwarding thunk for a constructor shall use the same initialization syntax that is used in the expression requirement that corresponds to it.

[14] Narrowing conversions and constructors taking instances of `std::initializer_list` are the most often responsible for this behavior.

If both syntaxes shall be valid for initializing a concept-erased object, then *two* expression requirements have to be written. Alternatively, a single signature requirement for constructors can be used, which eventually results in the generation of two erasing functions: one for regular initialization (unless the constructor is taking an instance of `std::initializer_list`) and one for brace-initialization.

Finally, conversion operators are treated just like regular functions, except for the fact that the compiler shall prohibit the use of erased explicit conversion operators in contexts that do not allow them (e.g. conversion to `bool` outside of an `if` statement).

5.6. Limitations on expression requirements

Sections 5.1 and 5.2 show how to extract the erasing signature and the definition of the forwarding thunk associated to a given expression requirement (or compound requirement) consisting of either a single free function call expression or a single member function call expression.

The Concepts TS (see [6]) does not place any restriction on the kind of expression that can appear in a compound requirement. For instance, definitions such as the following are perfectly valid:

```
template<typename T>
concept bool Foo = requires(T x)
{
    { x.bar(x.fizz() + x.buzz()) } -> double;
};
```

This makes sense, because all the compiler has to do in order to check whether a type models a concept is to try and compile the corresponding expression.

However, in the dynamic context the compiler has to generate an erasing interface from that expression. What should be the type of the only parameter (ignoring the implicit `this` pointer) in the erasing signature of `bar()`? As mentioned in Section 5.4, forwarding thunks cannot be templates, but we cannot assume any specific type either. Also, what should be the return types in the erasing signatures of `fizz()` and `buzz()`? All we can assume is that they are addable, i.e. that they model concepts which are bound by some expression requirement involving their sum.

The complexity of implicitly defining anonymous concepts when generating erasing interfaces just for the sake of supporting arbitrary expressions seems to not be worth the extra complexity. Therefore, concepts used virtually can only include expression requirements consisting of a single top-level (free or member) function call. Attempts to use concepts such as `Foo` above virtually shall result in a compiler error.

5.7. Associated types

Back in Section 4.6 we introduced an example of how associated type requirements could be useful in the context of DGP. Back then, we introduced two concepts, `DevicePart` and `Device`:

```
template<typename T>
concept bool DevicePart = requires()
{
    T::T(std::string);
    auto T::get_name() const -> std::string;
    requires SemiRegular<T>;
    // ...
};

template<typename T>
concept bool Device = requires()
{
    typename T::PartType;
    requires DevicePart<typename T::PartType>;
    auto T::add_part(PartType) -> void;
    // ...
};
```

We also defined models for the above concepts, respectively `Joint` and `Robot`:

```
struct Joint
{
    Joint(std::string);
    auto get_name() const -> std::string;
    // ...
};

struct Robot
{
    using PartType = Joint;
    auto add_part(Joint) -> void;
    // ...
};
```

Finally, we had a function creating a new part for a given Device argument and adding it to it:

```
void add_new_part(virtual Device& d, std::string name)
{
    Device::PartType p{std::move(name)};
    d.add_part(p); // THE COMPILER KNOWS THAT p HAS THE CORRECT TYPE
}
```

We can now show how associated erased types can be realized. First of all, type requirements become part of a concept's erasing interface, which means corresponding entries appear in instantiation tables for a given concept. However, unlike entries for function calls – which are pointers to forwarding thunks – these entries contain pointers to *other instantiation tables*.

This allows the compiler to navigate associations between types and correctly dispatch function calls at run-time. Figure 8 below illustrates how `itable(Robot, Device)` contains an entry for the associated `DevicePart` type pointing `itable(Joint, DevicePart)`:

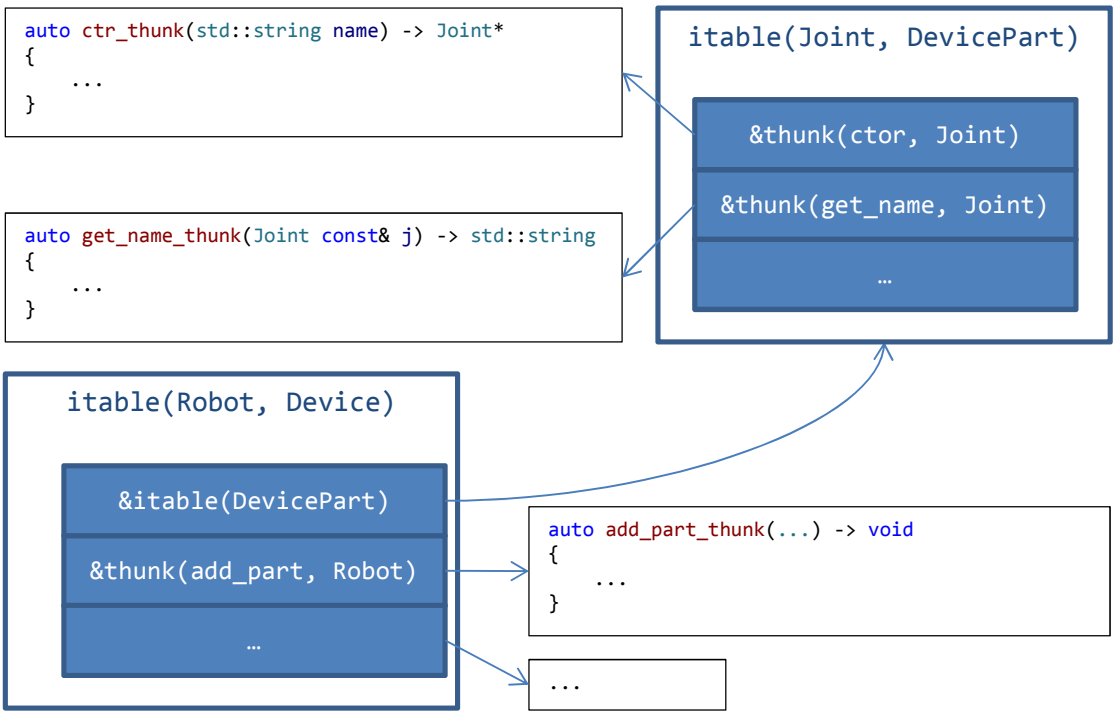

Figure 8 - Links between instantiation tables realize associated types.

Clearly, in order for this to be feasible, associated type requirements must be accompanied by an instantiation requirement that specifies what concept the associated type shall model. As I wrote in Section 4.6, in fact, the absence of such a requirement would force the compiler to treat the associated type as a `Void`, preventing pretty much anything to be done with the erased object other than binding it to some reference (see Section 4.2 for more information on `Void`).

5.8. Casting

When discussing degenerate erasure and the `Void` virtual concept back in Section 4.2, I made use of `dynamic_cast` to safely cast a concept-erased object down to its (known) actual type:

```
void foo(Void& x)
{
    auto i = dynamic_cast<int>(x); // THROWS std::bad_cast ON FAILURE

    // ...
}
```

Although the behavior is intuitive, the reader might have spotted a potential problem here: `dynamic_cast` only works with polymorphic types, i.e. types that have at least one virtual function – RTTI is not available for non-polymorphic types. However, `int` is not a polymorphic type, so how can this be made to work? (More generally, how can we apply `dynamic_cast` to an erased object which isn't guaranteed to be of a polymorphic type?)

The answer is that although `int` is not a polymorphic type, the moral equivalent of a virtual table for that type is accessible to the compiler in the form of `int`'s instantiation table for `Void`; the instantiation table can contain entries for RTTI just like a regular virtual table does, except instantiation tables are available for *any* type that happens to be erased through a virtual concept, including non-polymorphic ones.

The details of how this information is laid out in the instantiation table are outside the scope of the present proposal, but it is easy to imagine that the implementation of virtual tables for the given platform can be borrowed and applied to instantiation tables too.

Since type information is erased, a `static_cast` operation cannot be applied to a concept variable: the compiler cannot know, at compile-time, what offset shall be applied to the erased object pointer.[15]

A `reinterpret_cast` operation, on the other hand, is always possible, but the target type has to be identical to the static type of the object from which the concept variable was initialized – otherwise, the behavior of the program is undefined.

[15] Even if the cast does not involve pointer types, realizing value semantics for virtual concepts involves, in the general case, dynamic allocations and – therefore – requires manipulating object pointers under the hood. This will be further discussed in Section 6.2.

[WORK IN PROGRESS]

Bibliography

[1] P. Wadler, "Expression problem," [Online]. Available: http://en.wikipedia.org/wiki/Expression_problem.

[2] Adobe, "Runtime Concepts - Proxy Dilemma," [Online]. Available: http://stlab.adobe.com/wiki/index.php/Runtime_Concepts#The_Proxy_Dilemma.

[3] R. C. Martin, "The Dependency Inversion Principle," [Online]. Available: http://www.objectmentor.com/resources/articles/dip.pdf.

[4] M. Marcus, J. Järvi and S. Parent, "Runtime Polymorphic Generic Programming - Mixing Objects and Concepts in ConceptC++".

[5] S. Watanabe, "Boost.TypeErasure," [Online]. Available: http://www.boost.org/doc/libs/1_56_0/doc/html/boost_typeerasure.html.

[6] A. Sutton, *Technical Specification: Concepts,* 2014.

[7] A. Krzemieński, "Type erasure — Part I," [Online]. Available: http://akrzemi1.wordpress.com/2013/11/18/type-erasure-part-i/.

[8] Adobe, "Adobe.Poly," [Online]. Available: http://stlab.adobe.com/group__poly__related.html.

[9] P. Jahkola, "Poly. Solving The Expression Problem in C++11," [Online]. Available: https://github.com/pyrtsa/poly.

[10] J. Turkanis, "Boost.Interfaces," [Online]. Available: http://www.coderage.com/interfaces/.

[11] "Boost C++ Libraries," [Online]. Available: http://www.boost.org/.

[12] G. Baumgartner and V. F. Russo, "Signatures: A Language Extension for Improving Type Abstraction and Subtype Polymorphism in C++," *Computer Science Technical Report,* 1995.

[13] GNU, "Extensions to the C++ Language," [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc_5.html#SEC112.

[14] V. F. R. Gerald Baumgartner, "Implementing Signatures for C++," *ACM Transactions on Programming Languages and Systems,* vol. 19, no. 1, pp. 153-187, 1997.

[15] Adobe, "Adobe Source Libraries," [Online]. Available: http://stlab.adobe.com/group__asl__overview.html.

[16] S. Parent, "Concept-Based Runtime Polymorphism," 17 5 2007. [Online]. Available: http://stlab.adobe.com/wiki/images/c/c9/Boost_poly.pdf.

[17] P. Pirkelbauer, S. Parent, M. Marcus and B. Stroustrup, "Runtime Concepts for the C++ Standard Template Library," in *23rd ACM symposium on applied*, Fortaleza, Ceara, Brazil, 2008.

[18] S. Parent, "Inheritance Is The Base Class of Evil," 6 12 2013. [Online]. Available: http://www.google.com/url?q=http%3A%2F%2Fchannel9.msdn.com%2FEvents%2FGoingNative%2F2013%2FInheritance-Is-The-Base-Class-of-Evil&sa=D&sntz=1&usg=AFQjCNHVlpf_jVIeF8I3kBy0FblJ9lq5gA.

[19] S. Watanabe, "Syntax Limitations of Boost.TypeErasure," [Online]. Available: http://www.boost.org/doc/libs/1_56_0/doc/html/boost_typeerasure/any.html#boost_typeerasure.any.limit.

[20] S. Watanabe, "Defining Custom Concepts with Boost.TypeErasure," [Online]. Available: http://www.boost.org/doc/libs/1_56_0/doc/html/boost_typeerasure/concept.html#boost_typeerasure.concept.custom.

[21] S. Watanabe, "Overloading with Boost.TypeErasure," [Online]. Available: http://www.boost.org/doc/libs/1_56_0/doc/html/boost_typeerasure/concept.html#boost_typeerasure.concept.overload.

[22] C. Diggins, "Object Oriented Template Library (OOTL) Version 0.1," [Online]. Available: http://www.artima.com/weblogs/viewpost.jsp?thread=81724.

[23] C. Diggins, "Smart Interface Pointers in Boost.Interfaces," [Online]. Available: http://www.coderage.com/interfaces/libs/interfaces/doc/index.html.

[24] Z. Laine, "Pragmatic Type Erasure: Solving OOP Problems w/ Elegant Design Pattern," [Online]. Available: https://www.youtube.com/watch?v=0I0FD3N5cgM.

[25] B. Stroustrup, "Make Simple Tasks Simple!," 09 2014. [Online]. Available: https://www.youtube.com/watch?v=nesCaocNjtQ.

[26] A. Krzemieński, "Value Semantics," 3 2 2012. [Online]. Available: http://akrzemi1.wordpress.com/2012/02/03/value-semantics/.

[27] S. Parent, "C++ Seasoning," 2013. [Online]. Available: http://channel9.msdn.com/Events/GoingNative/2013/Cpp-Seasoning.

[28] D. Gregor, J. Jarvi, J. Siek, B. Stroustrup, G. D. Reis and A. Lumsdaine, "Concepts: Linguistic Support for Generic Programming in C++," in *OOPSLA'06*, Portland, 2006.

[29] K. Henney, "Boost.Any," 2001. [Online]. Available: http://www.boost.org/doc/libs/1_56_0/doc/html/any.html.

[30] B. Stroustrup, "Simplifying the use of concepts," 21 06 2009. [Online]. Available: http://www.open-std.org/Jtc1/sc22/wg21/docs/papers/2009/n2906.pdf.

[31] D. Gregor, B. Stroustrup, J. Widman and J. Siek, "N2617 - Proposed Wording for Concepts (Revision 5)," 19 05 2008. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2617.pdf.

[32] R. Johnson, J. Vlissides, R. Helm and E. Gamma, Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education, 1994.

[33] M. Lipovaca, "Learn You A Haskell," [Online]. Available: http://learnyouahaskell.com/types-and-typeclasses.

[34] R. Lämmel, "Advanced Functional Programming - The Expression Problem," 10 8 2010. [Online]. Available: http://channel9.msdn.com/Shows/Going+Deep/C9-Lectures-Dr-Ralf-Laemmel-Advanced-Functional-Programming-The-Expression-Problem.

[35] P. Walton, "A Gentle Introduction to Traits in Rust," 8 8 2012. [Online]. Available: http://pcwalton.github.io/blog/2012/08/08/a-gentle-introduction-to-traits-in-rust/.

[36] B. Milewski, "C++ Concepts: a Postmortem," [Online]. Available: http://bartoszmilewski.com/2010/06/24/c-concepts-a-postmortem/.

[37] A. Sutton, 9 2014. [Online]. Available: https://www.youtube.com/watch?v=NZeTAnW5LL0.

[38] A. Sutton, "The Origin Library," [Online]. Available: https://github.com/asutton/origin.