Semantics-Driven Genericity: A Sequel to the Static C++ Object-Oriented Programming Paradigm (SCOOP 2)

Thierry Géraud, Roland Levillain

EPITA Research and Development Laboratory (LRDE)
14-16, rue Voltaire, FR-94276 Le Kremlin-Bicêtre Cedex, France
thierry.geraud@lrde.epita.fr, roland.levillain@lrde.epita.fr
Web Site: http://www.lrde.epita.fr

Abstract. Classical (unbounded) genericity in C++03 defines the interactions between generic data types and algorithms in terms of *concepts*. Concepts define the requirements over a type (or a parameter) by expressing constraints on its methods and dependent types (`typedef`s). The upcoming C++0x standard will promote concepts from abstract entities (not directly enforced by the tools) to language constructs, enabling compilers and tools to perform additional checks on generic constructs as well as enabling new features (e.g., concept-based overloading). Most modern languages support this notion of signature on generic types. However, generic types built on other types and relying on concepts—to both ensure type conformance and drive code specialization—restrain the interface and the implementation of the newly created type: specific methods and associated types not mentioned in the concept will not be part of the new type. The paradigm of concept-based genericity lacks the required semantics to transform types while retaining or adapting their intrinsic capabilities. We present a new form of semantically-enriched genericity allowing static, generic type transformations through a simple form of type introspection based on type metadata called *properties*. This approach relies on a new Static C++ Object-Oriented Programming (Scoop) paradigm, and is adapted to the creation of generic and efficient libraries, especially in the field of scientific computing. Our proposal uses a metaprogramming facility built into a C++ library called Static, and doesn't require any language extension nor additional processing (preprocessor or transformation tool).

1 Introduction

In the context of software library design for numerical scientific computing, we want to meet two major objectives at the same time: *efficiency* because of the large data sets to be processed, and *genericity* since data can have multiple different types. Many libraries fulfilling such aims have been designed and developed with the C++ language [1], which has proved to be an appropriate tool [2,3,4,5]. The way *abstractions* can be handled in scientific libraries is a subject

of prime importance: practitioners mainly think about the entities of their domain in terms of abstractions, and consider implementation aspects in second place. A great benefit brought by the C++ language comes from the fact that it features multiple paradigms; as a consequence, it offers several solutions to design well-grounded scientific libraries where implementation classes are tightly related to properly defined abstractions.

The Generic Programming (GP) paradigm is classically presented as a means to achieve both efficiency and genericity in scientific libraries, leading to a strong decoupling between:

- *structures*, the different kinds of containers offered to represent data sets;
- *values*, the different types of elements that can be stored in structures;
- and *algorithms*, some non-elementary operations that are expected to run over data sets.

Schematically, providing S structure types, V value types, and A generic algorithms—i.e $S + V + A$ entities—a library features $D = S \times V$ different data sets (input types) and $D \times A$ possible processing routines. Yet, another category of entities happens to be useful:

- *data modifiers*, such as views, adapters, decorators, wrappers, or any transformation applied on a single or several data sets to provide the user with "other" data sets than the primitive ones.

The obvious requirements over such modifiers are those we already have for the other entities. They shall not penalize performance at run-time. They also have to be generic, that is, they shall apply to as many data sets as possible. Therefore with M modifiers, the number of expressible data types becomes $D^{(M^*)}$. In addition to the efficiency and genericity requirements, their main property is the following: *a modifier type is written once while being able to modify the data types it applies to, no matter the specificity of the original data types.*

Some examples of such modifiers already exist. For instance, the Boost Iterator Library [6] proposes a collection of "specialized adaptors." Though they do not directly apply on containers (data sets) but on iterators (data access), they affect the behavior of algorithms just like if the nature of the containers had changed. All iterators do not share the exact same interface, so the adapter classes have to handle this variability; thus, tag types are introduced in order to discriminate between the different abstractions of iterators. Given an iterator of type `I`, its tag can be retrieved through `iterator_traits<I>::iterator_category`, that is, an associated type enclosed in a traits class. When the result is `random_access_iterator_tag`, we know that `I` satisfies the corresponding abstractions. An adapter class targeting the type `I` shall then rely on this piece of information to offer the appropriate interface and behavior.

In a generic library featuring many different abstractions, several of them being independent from some other ones, providing dedicated *modifier* classes on a per-case basis is clumsy. We really need a very general mechanism to easily design these classes whose main characteristic is that they should be highly adaptable.

To address these problems, we have extended the Static C++ Object-Oriented Programming (Scoop) paradigm presented in [7]. The need for this paradigm comes from the development of two numerical scientific computing libraries, namely Olena [8,9] for image processing and Vaucanson for finite automata manipulation [10,11]. Scoop 1 proposed an approach mixing classical Object-Oriented Programming (OOP) and GP, but no support to implement generic modifiers. This lack finds its origins in the limitation of existing paradigms. To allow the creation of generic modifiers, we propose to reverse classical OOP, by introducing the idea of *properties* within the second version of Scoop. While OOP is a top-down process (upper classes imposes type constraints on lower classes), properties spawns a bottom-up mechanism: concrete classes express a kind of signature from their properties, which is used to select the abstractions they conform to, and possibly, retrieve automatically some implementations based on user-written rules.

This paper is structured as follows: first, we introduce the idea of properties by studying existing GP-based paradigms in Section 2. Then, we present the Scoop 2 paradigm (Section 3), and a component to apply it to generic libraries, Static (Section 4). We compare our approach to existing work in Section 5. Section 6 concludes.

2 The need for properties

This section compares various approaches to express generic programming in the C++ language, and draws a first conclusion on the current GP techniques with respect to the implementation of generic modifiers: these paradigms fail to meet our expectations. A proposition introducing the concept of properties is then presented.

2.1 Classic genericity: C++03

In contrast with OOP, generic programming does not require class inheritance but heavily relies on parametric polymorphism thanks to the C++ `template` keyword. Libraries built with GP [2,5,12,4] are efficient since the run-time cost of `virtual` methods is avoided; furthermore, methods code can be inlined which allows the compiler for extra optimizations. Abstraction interfaces are not mapped into source code but are described in documentation. The relationship between implementations and abstractions is *implicit*: an implementation class just has to provide what is required by its abstractions.

2.2 Concept-based genericity: C++0x

With the evolution of the C++ Standard, programmers will benefit from the introduction of the `concept` keyword into the upcoming C++0x proposal [13] in order to materialize into source code a set of requirements over types, a feature present in popular modern languages [14]. For instance, an abstraction

interface could be mapped into a C++0x concept and, given a generic algorithm—a parametrized function—this concept could be used to define a constraint over eligible parameters. This evolution thus preserves all the benefits from "classical" generic programming while enhancing code expressiveness, program safety, and design capabilities. With the advent of concepts comes the notion of where-clauses, a language construct to enforce a set of requirements using concepts on a type. Where-clauses allow a new form of function overloading based on concepts, which greatly enhances algorithm specialization [15].

Though we will have to wait for the next C++ standard to be able to manipulate concepts as entities of the language, there are already tools partially reifying concepts for C++03 [16]. They help to enforce early conformance of types to the concepts they model, and provide better error messages. Likewise, a form of concept-based overloading is possible with C++03 [17]. However, these techniques have less expressiveness and are less robust than their future C++0x counterparts.

2.3 Mixing Genericity and OOP: SCOOP 1 (using C++03)

In [7] we proposed a solution to translate a classical, hence dynamic, OOP class hierarchy into a *static* one. Put shortly, SCOOP 1 is a mix between OOP and GP in order to take advantage of both worlds. From OOP it borrows inheritance so that interfaces are explicitly defined as classes; SCOOP then allows for code factoring, even if some definitions (methods or associated types) are still unknown and deferred to sub-classes. From GP it keeps efficiency since all classes are parametrized and statically resolved. Technically speaking SCOOP 1 is based on a generalization of the Curiously Recurring Template Pattern [18]. The main difference between SCOOP 1 and C++0x GP is that the relationship between a concrete class and an abstraction is *explicit* with the former (as in OOP), whereas it is *implicit* with the latter.

2.4 Limitations of these approaches

The main difficulty of implementing generic modifiers comes from handling specificity. To illustrate our point, let us consider a modifier class that acts as an adapter. For instance, in the C++ standard library the class template `std::queue` is an adapter over an underlying sequence. Some eligible type `C` for the sequence can be `std::list<T>` (doubly linked list) or `std::deque<T>` (double-ended queue), `T` being the queue element type. Let us imagine that this queue type *is a modifier*. Under this assumption, its interface can provide a random access facility to queue elements if and only if `C` already features this same facility. That is the case when `C` is a deque but not when it is a list. The interface of `queue<T,C>`, and its related code, thus *depends upon the interface of its parameter* `C`. With current C++, `std::queue<T>` is not a modifier, and it masks the interface of `T`. This is an artifact of classical genericity that we call *restraining genericity*.

The first restriction imposed by modifiers is that they prevent the use of inheritance as a means to build upon an existing type. Modifiers are much more generic than sole decorators or proxies: they shall allow any transformation from a generic (and equipped) type, including *changing its interface* (and therefore its implementation). Let us consider an example from the domain of image processing: if we create a transformation flattening a 3-dimensional image to a 2-D one (i.e., a generic modifier reducing the dimension of a generic 3-D image type), we need to adjust the interface of the image type with respect to the type of points, the type of the grid, etc. (i.e, translate them to their 2-D counterparts).

This example shows that Classical GP is too limited to implement modifiers. Moreover, it seems the new features of C++0x evoked in section 2.2 will not suffice to implement modifiers directly either: using a type `T` conforming to a given concept `C` to create a new type `U` will not take into account the specific part of the interface of `T` absent from `C` if `T` models in fact a *refinement* of `C` (i.e., a sub-concept). Likewise, Scoop 1 type transformations are only limited to a fixed type signature. In any case, there is no automatic way to retrieve default implementation while transforming types, based on a given interface (no static introspection mechanism).

The main idea of this paper is that type transformations should be available as a GP feature, and require an *extended generic programming paradigm* to be implemented.

More generally, though templates offer useful static metaprogramming tools in C++, the language itself lacks a general-purpose static meta-programming facility, like complete static introspection or a meta-object protocol [19].

Some modern languages like Java and C# offer a *dynamic* introspection mechanism, but this is a different service: we are looking for a static means to interact with the compilation process. For instance, the C++ language does not provide a actual means to inspect the `typedef`s of a class, though this information is required to create non-trivial type transformations.

2.5 Semantics-driven genericity

Modifiers should therefore perform some kind of static introspection on the transformed type, to acquire information on its interface and its implementation. This work will be performed by metaprogramming algorithms [20], so the information has to be encoded as types.

One cannot easily extract information from a C++ program without an external tool, in order to perform type transformations. Thus, the programmer has to provide the compiler with some information characterizing the very nature of the type to be transformed. With well known template-based static metaprogramming techniques, the compiler will be able to both *fetch* information form the type (rather than *inspect* it) and even rely on existing implementations through a delegation mechanism. The idea is that abstractions should be able to express the requirements of type transformations on data using *properties*, that is, parameters depending on the exact type of the implementation class. They should therefore be part of the definition of concrete classes of the hierarchy.

This separation lays down a first issue in C++, as it introduces a recursion in the definition of a class: a concrete class needs to build its super classes first; but these super classes, also used to represent the abstractions modeled by the concrete class, depend on types defined in the concrete class.

We must first recall that the solution cannot solely rely on direct inheritance from the transformed type: as the semantics of the transformation does not always allow it. For instance, given a 3-D image of type `T`, it would be tempting to define a slice (a 2-D image) of an object of type `T` as an instance of a subclass of `T`: a slice of a 3-D image is indeed a 2-D image; a 2-D image, however *is not* a 3-D image, and has a different interface. Nevertheless, inheritance is the only usual way to have a type `U` automatically retrieve (a part or all of) the implementation of a type `T` in C++. Therefore we propose a programming paradigm based on

- a split pattern, with abstractions expressing concepts using *abstract types* (depending on the exact type of the model) on the one hand; and implementations classes on the other hand;
- a set of properties attached to any concrete class of the hierarchy, used to give a "value" to the expected types in the signature of routines of abstractions. Properties add semantics from the target domain (e.g., image processing) to data types, and are used to automatically drive the inheritance relationships from concrete classes to the right abstractions;
- a delegation mechanism to automatically retrieve implementation from a *delegatee* to create generic type transformations.

These ideas are the heart of the Scoop 2 paradigm, which enriches the semantics of a library: qualifying a type with is properties and using them to retrieve both interfaces and implementations statically augment the expressiveness of the language.

3 The SCOOP 2 paradigm

Scoop (standing for *Static C++ Object-Oriented Programming*) is a programming paradigm that addresses common issues raised in the context of generic and efficient programming. Its first version has been described by Burrus *et al.* [7], and was used to build the Olena generic image processing library [8], version 0.10. This section presents the second version of the paradigm, which has been design for the forthcoming Olena library [21].

Scoop 2 has been developed primarily to fulfill the needs of designers of scientific computing libraries (especially Olena and Vaucanson), where genericity and run-time efficiency are fundamental.

Because the paradigm has an impact on the design of the software, Scoop 2 aims mainly at building new libraries. However, one can use it within an existing code base, either non intrusively (by wrapping existing data structures), or intrusively, by modifying existing code. We have successfully applied the first approach to a subset of the standard C++ library (some containers and algorithms), as a proof-of-concept of Scoop 2 [22].

One of the main advantages of Scoop 2 is that this paradigm is expressed directly in the host language (C++ to be specific), hence it doesn't require any software tool other than a compiler complying with the ISO/IEC C++ Standard [1]. Nevertheless, to both formalize the implementation of the paradigm and factor out the development among clients, we propose a library providing a part of the facility used to apply the paradigm: types and metaprogramming algorithms. This component is called Static, because most of its code is used to create so-called *static class hierarchies*, where the exact type of every object is known at compile-time, replacing dynamic dispatch of method calls with static ones and allowing some form of static class introspection. Static is built upon Metalic, another library delivering basic C++ template metaprogramming tools (manipulation of types as values, logic on types, static `if` and `switch` statements on types expression, `typedef` lookup, etc.). Metalic shares similar goals with Loki [23] and the Boost MPL library [24,25], and could be replaced by these libraries in a future implementation of Static. Notably, the atomic conformance check technique (for typedef introspection and subtype checks), based on SFINAE and `sizeof`, is from Loki [23, section 2.9]. Section 4 is dedicated to Static. Please note that unexplained elements from examples illustrating Static uses (figures 4, 6 and 8) are addressed in later parts of the article to enhance readability. Their object is basically to show how Static client code reads, as these samples are actual code.

3.1 Organizational Considerations

The design of Scoop determines the roles of people taking part in a Scoop-based library project. We can classify these actors in a taxonomy of four categories.

Simple Users For this kind of actor, the components provided by the library (data types and algorithms) are sufficient to solve a problem of the applicative domain. A Simple User thus just assembles components and knows nothing about the library internals.

Designers This actor designs new algorithms, so he extends the number of algorithms (A).

Providers This actor provides the library with new data structures, hence he increases the number of data structures (S), and possibly the number of value types (V).

Architects At the opposite of Simple Users, this category of actors is mainly concerned by the library internals.

A fifth category, maintainers, contains people in charge of the low-level components above which the library is built. As they are not directly related to the library, their role will not be covered in this section.

The programming skills expected from an actor to enter one of those categories usually go increasingly from Simple Users to Architects; fortunately, the size of the population of those categories follows an opposite trend. However,

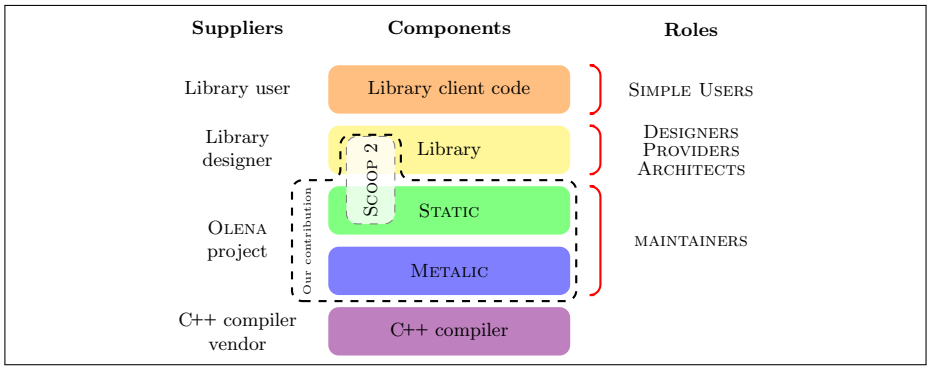

Fig. 1. Components and actors in a Scoop 2-based project. The vertical layout represents the dependencies between components: a component located on top of another depends on the latter.

as a matter of fact in the context of scientific libraries, a lot of people are involved in designing algorithms and there are a lot more Designers than Providers. A quality criterion for libraries is that the compliance between algorithms and data should be maximal. Having also in mind the notions of reusability and extensibility, this criterion has two consequences that respectively fall in the hands of Designers and Providers. Figure 1 summarizes the roles of each user.

3.2 Key concepts of the design

Scoop 2 is to be used to build libraries composed of generic algorithms (as function templates) and generic data structure (as class templates). Conceptually, Scoop 2 pays more attention to the *data* aspect than many other generic library designs. Indeed, a lot of these libraries draw their generic orientation from their algorithms, and data structures are often considered as mere models of the concepts used by these algorithms. Although these libraries provide means to extend data structures through adapters or wrappers [2,12,26], these approaches benefit little or nothing from the generic strategy (from the reusability and factoring point of view): adapters and wrappers are generally not meant to be combined.

3.3 Data structures

The developer of the library shall organize the various entities of the domain in *categories*. For instance, a generic image processing library would comprise categories such as image, point, neighborhood, etc. For each category of data structure, Scoop 2 invites the Designer to create two class hierarchies: one for the abstraction(s) (reified as "concepts" in C++0x) and one for the implementation(s). Following the previous example, corresponding abstractions for

the category image would be Image2d, Image3d, BinaryImage, ColorImage, MutableImage, etc.

Hierarchy of abstractions Concepts are expressed as abstract classes in the paradigm. Although the upcoming C++ Standard is to propose concepts as actual language constructs [13], these are not powerful enough to fully support SCOOP 2. "Concepts-as-classes" is notably a requirement of the delegation mechanism of the paradigm (see 3.3). The hierarchy is used as a means to organize the concepts according to their acquaintances (refinement, orthogonality, exclusion). The generalization relationship mimics a *refinement* in this hierarchy. Abstractions are detailed in 3.3.

Hierarchy of implementations Actual data structures are organized in class hierarchies, which serve to factor implementations. Data types are either *primary* (not relying on any other data type of the same kind) or *composed* (aggregating one or more objects of the same kind). In the latter case, the paradigm uses a powerful delegation mechanism allowing type transformations and retaining the semantics of the original type, a claim brought up in Section 2. Section 3.3 gives more insight on implementation hierarchies.

Both hierarchies can be extended independently. This is immediate for implementations, thanks to inheritance—and because properties are automatically inherited, thanks to STATIC (see section 4.3). Inheritance allows an extension from the *bottom* of the hierarchy, which fits with the location of implementations within the whole class diagram. However, things get a little trickier when it comes to abstractions, as existing implementations might already inherit from them. Extending an abstraction hierarchy *a posteriori* requires some equipment to allow a form of declarative inheritance and have implicit links between implementation and abstraction be extended non intrusively.

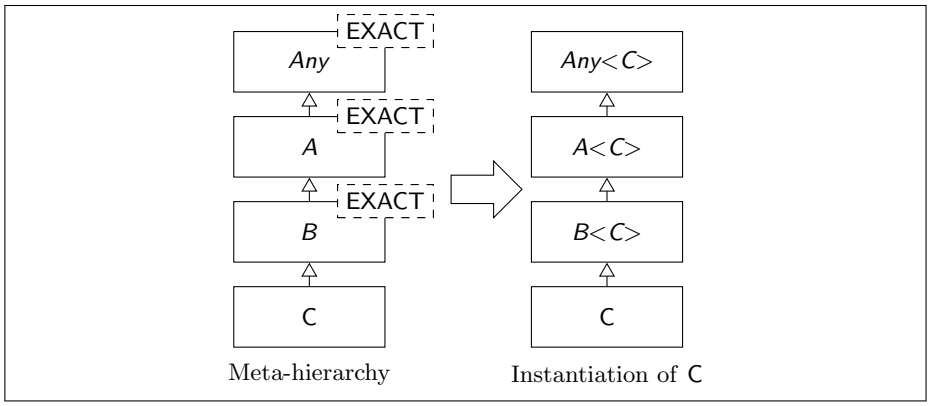

Fig. 2. An example of Static hierarchy unfolding.

Implementation hierarchies Implementation classes are templates organized in hierarchies using only single inheritance (though the paradigm can be extended to support multiple inheritance). Class hierarchies follow a generalization of the Curiously Recurring Template Pattern (CRTP) [18]. In the Generalized Curiously Recurring Template Pattern (GCRTP) [7], each derived class pass its type as parameter to its super class recursively, so that every super class knows the exact type of the object. (see Figure 2). The idea of GCRTP was previously evoked by Veldhuizen [20].

Static comes with a top-level class `Any` to make the integration of the GCRTP easier, and a routine `exact` returning a pointer with the exact type of an object, allowing static dispatch of method calls. GCRTP-based hierarchies are known as *static hierarchies*. Figure 3 gives an example of such a hierarchy. The implementation of `Any` is addressed in section 4.2.

```
// Abstractions.
template <typename Exact>
struct A : public Any<Exact> {
  // Static dispatch.
  void m () { exact(this)->impl_m (); }
};

template <typename Exact>
struct B : public A<Exact> {
  // Static dispatch.
  void n () { exact(this)->impl_n (); }
};

// Implementation.
struct C : public B<C> {
  // Implementations of m() and n().
  void impl_m () { /* ... */ }
  void impl_n () { /* ... */ }
};
```

Fig. 3. An example of static hierarchy.

The paradigm relies on *properties*, which determine the nature of data structures of the library and to which abstractions they conform—or, in other words, which concepts they model. The link between implementations and abstractions can be either explicit (thanks to traditional inheritance; this approach is similar to using a `concept_map` in C++0x) or implicit (computed at compile time from the properties of the structure). Section 3.3 elaborates on this subject. These properties are expressed for each class as *virtual types* (see 3.3). In the following, we call Scoop *classes* the types that are part of a Scoop static hierarchy. Figure 4 shows some Scoop classes from the Olena library.

Abstraction hierarchies Scoop 2 uses classes to reify abstractions, the entities which express the concepts of the library. These classes are akin to Java interfaces, in the sense that

```
// image_base.
// -----------
template<typename Exact> class image_base;

template<typename Exact>
struct super_trait_< image_base<Exact> > { typedef top<Exact> ret; };

template<typename Exact>
struct vtypes< image_base<Exact> > {
private:
  typedef stc_deferred(point) point_;
public:
  // Virtual type 'category' is used by top<Image> to
  // link image_base to the right abstraction(s).
  typedef stc::final< stc::is<Image> >      category;
  typedef stc::final<typename point_::grid> grid;
  typedef stc::final<point_>                psite;
};

template <typename Exact>
struct image_base
  // Implicit (computed) link between image_base
  // and its abstraction(s).
  : top<Exact> {
  // ...
protected:
  image_base () {};
};


// image2d.
// --------
template<typename T> class image2d;

// image2d inherits properties from image_base.
template<typename T>
struct super_trait_< image2d<T> > { typedef image_base< image2d<T> > ret; };

template<typename T>
struct vtypes< image2d<T> > {
  typedef point2d point;
  typedef T       value;
};

template <typename T>
class image2d : public image_base< image2d<T> > {
public:
  typedef image2d<T> self;
  typedef image_base<self> super;
  // Import some virtual types of image2d inside
  // the scope of the class.
  stc_using(point);
  stc_using(value);
  // Implementation of Image::operator().
  value impl_read(const point& p) const { /* ... */ }
  // ...
};
```

Fig. 4. Some Scoop 2 (implementation) classes for the category image from the Olena library (abridged). `stc_deferred` and `top` are explained in sections 4.3 and 4.4 respectively. As for `stc::is<Image>`, a common way to "tag" classes as belonging to a given a category is to set their virtual type `category` to `stc::is<Abs>`, where `Abs` is their topmost abstraction. `Image` is an abstraction presented in Figure 6.

– their member shall be generic routines or dispatchers;
– they shall carry no data (attributes);
– they shall define no virtual type, though they can make use of them (i.e., use virtual types that will be defined in implementation classes).

Abstractions are passed the exact type of the class being instantiated, like any super class in the GCRTP. Concrete types may fulfill several abstractions (through inheritance) to model several concepts, so as to uncouple the requirements of various algorithms and add statically dispatched multimethods to the library. Static multimethods were a part of Scoop 1, and are covered in [7]. Two (or several) concepts of a library can be

related through a refinement relationship The requirements of a concept can form a superset of another one, which translates to a relation of generalization (inheritance) between their corresponding abstractions. For example, a BinaryImage is a refinement of a LabelImage.

concurrent A given concrete class can logically model one of several concurrent concepts, because they are from the same domain (implying their corresponding abstractions belong to the same hierarchy). For instance, ColorImage and GrayLevelImage are concurrent abstractions.

orthogonal They express unrelated requirements (their abstractions belong to different hierarchies). As an example, Image2D and ColorImage are orthogonal abstractions.

The abstractions of our library are thus organized as *orthogonal hierarchies* (see Figure 5). Figure 6 gives an example of abstractions corresponding to the Scoop classes of Figure 4.

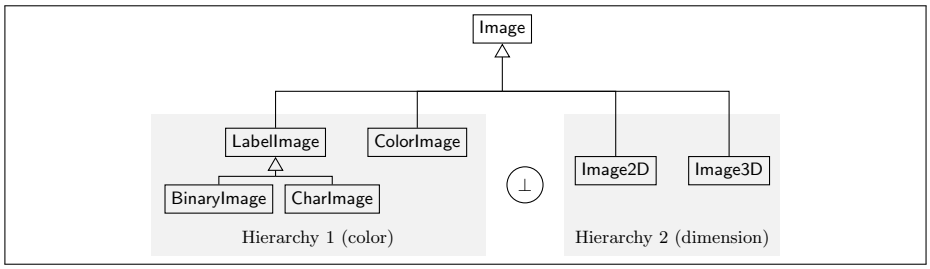

Fig. 5. An example of hierarchies of abstractions. Abstractions belonging to category image are all sub-abstractions of Image. Abstraction BinaryImage is a refinement of LabelImage. Though this is not enforced by any language construct, abstractions LabelImage (and its sub-abstractions) are semantically concurrent with ColorImage , as they are part of the same (logical) hierarchy. Likewise, abstractions of hierarchy 1 are orthogonal to abstractions of hierarchy 2.

```
// Abstractions.
// -------------
template <typename Exact>
struct Image : public virtual Any<Exact> {
  stc_typename(grid);   // Type of grid.
  stc_typename(point);  // Type of point.
  stc_typename(psite);  // Type of point site.
  stc_typename(value);  // Type of value.
  // Return the value at site P.
  value operator()(const psite& p) const {
    return exact(this)->impl_read(p);
  };
  // ...
protected:
  Image() {};
};


// Abstraction sub-hierarchy for value kind.
// -----------------------------------------
template <typename Exact>
struct LabelImage : public virtual Image<Exact> { /* ... */ };

template <typename Exact>
struct BinaryImage : public LabelImage<Exact> { /* ... */ };

template <typename Exact>
struct StringImage : public LabelImage<Exact> { /* ... */ };

template <typename Exact>
struct ColorImage : public virtual Image<Exact> { /* ... */ };

template <typename Exact>
struct DataImage : public virtual Image<Exact> { /* ... */ };


// Abstraction sub-hierarchy for dimension.
// ----------------------------------------
template <typename Exact>
struct Image1D : public virtual Image<Exact> { /* ... */ };

template <typename Exact>
struct Image2D : public virtual Image<Exact> { /* ... */ };

template <typename Exact>
struct Image3D : public virtual Image<Exact> { /* ... */ };

// Other abstraction sub-hierarchies
// ...
```

Fig. 6. Some abstractions for the category image from the Olena library (abridged). Note that there are no virtual type definition here, only uses of them.

Link between implementation and abstraction(s) Linking an implementation class to its abstraction(s) can either be *explicit* (manifest) or *implicit* (computed from the properties of the exact type of this class).

Explicit link In this case the link is an inheritance relationship between the implementation and the abstractions corresponding to the modeled concepts. As this type of link is explicitly written by the implementer, it is said to be "hard". It cannot be changed afterwards (other than by altering the definition of the implementation), which might break the extensibility of the library with respect to the abstractions and the type transformations that might be applied to the class.

Implicit link One of the new features of Scoop 2 is the ability to express the link between an implementation class of a given category and its abstractions as *rules on the properties* of this class, called *selectors*. For instance, a class ima from the category image whose properties contains a virtual type `grid` (type of grid) set to `grid2d`, shall inherit from the abstraction Image2D. In addition, if this class has two virtual types `psite` (type of point site) and `point` (type of point) having the same value (`point2d`), ima shall inherit from PointWiseAccessibleImage2D instead, which refines both the concepts Image2D and PointWiseAccessibleImage.

```
Stc-Is-A(source, target, abstraction)
1  val ← Find(source, target)
2  return Mlc-Is-A(val, abstraction)

Image-Value-Kind-Selector(source)
1  switch
2    case Oln-Is-Binary(Find(source, value)) : return BinaryImage
3    case Oln-Is-String(Find(source, value)) : return StringImage
4    case Oln-Is-Label(Find(source, value)) : return LabelImage
5    case Oln-Is-Color(Find(source, value)) : return ColorImage
6    case default : return DataImage

Image-Dimension-Selector(source)
1  switch
2    case Stc-Is-A(source, grid, Grid1d) : return Image1D
3    case Stc-Is-A(source, grid, Grid2d) : return Image2D
4    case Stc-Is-A(source, grid, Grid3d) : return Image3D
```

Algorithm 1: A selector used in the Olena library for the category image, written as pseudo-algorithm. The C++ template metaprogramming version is given in Figure 7. The algorithm Find is detailed in section 4.3. Mlc-Is-A is a metaprogramming algorithm provided by Metalic, while Oln-Is-Binary, Oln-Is-String, Oln-Is-Label and Oln-Is-Color are provided by Olena.

STATIC provides a means to express this semantics-driven (or property-based) inheritance. The system is based on a declarative mechanism using METALIC's meta-`switch`es on types. For each orthogonal trait of a category in the library (e.g., for a category `image`: image dimension, kind of value held by pixels), the DESIGNER writes cases as specializations of the meta-`case` statement. As METALIC's `case` statements are numbered template specializations, ARCHITECTS of the library or PROVIDERS of algorithms can extend the set of selectors non-intrusively by supplying additional `case`s, (though the order of the cases cannot be changed in the current implementation). Some rules linking the implementations of Figure 4 to the abstractions of Figure 6 are given in Algorithm 1. Figure 7 shows the C++ implementation of the corresponding selector.

Contrary to the concept-based approach which relies on where-clauses and either structural conformance or explicit modeling rules (through `concept_map`s) [13], the SCOOP 2 approach uses a property-based template metaprogramming algorithm to express the rules linking implementations to abstractions. This feature allows the library DESIGNER to write generic types based on other types of the library (modifiers, which we also call *morphers*), while considering their intrinsic specificity (see 3.3). At the present time this approach seems intractable using solely C++0x's concepts. The mechanism is explained in section 4.4.

Virtual Types The cornerstone of our proposal is the use of virtual types to express properties. Semantically, a virtual type of a SCOOP class plays the role of a polymorphic associated type of this class (or a virtual `typedef` in the C++ terminology). A virtual type is a type declaration or definition whose "value" is an actual C++ type. Like virtual member functions, virtual types are inherited and can be overridden in derived classes. Such redefinitions applies to the super classes, like polymorphic methods in the Inclusion Paradigm; that is, the value of a virtual type is always computed from the exact type.

A virtual type can be set **Abstract** in a class, i.e. declared with no definition; it might be then redefined in subclasses to be given an actual "value". Using a SCOOP class having at least one abstract virtual type is considered invalid in the SCOOP 2 paradigm. While STATIC cannot automatically enforce this check when instantiating the type, it provides some equipment macros to check the soundness of instantiated classes afterwards. Otherwise, (previously unchecked) invalid uses are caught the first time the abstract virtual type is used (see the algorithms FIND and CHECK in section 4.3). A virtual type can also be tagged **Final**, with the same meaning as its homonym in the Java programming language, i.e. to forbid redefinitions of this virtual type in subclasses.

Virtual types are used to define associated types like traits in most C++ generic libraries. Generic algorithms make use of these virtual types to abstract and generalize their definition ensuring maximum reusability [27]. *Defining* virtual types is very similar to defining traits, although SCOOP 2 cannot just rely on traits to *use* virtual types. It require metaprogramming algorithms to handle specific features like virtual type inheritance, **Abstract**, **Final** and delegation (see 3.3). The implementation of virtual types is discussed in section 4.3.

```
// Selector #1: value kind.
typedef selector<Image, 1> Image_value_kind;

template <typename Exact>
struct case_< Image_value_kind, Exact, 1 >
  : where_< value::is_binary<stc_get_type(value)> > {
  typedef BinaryImage<Exact> ret;
};
template <typename Exact>
struct case_< Image_value_kind, Exact, 2 >
  : where_< value::is_string<stc_get_type(value)> > {
  typedef StringImage<Exact> ret;
};
template <typename Exact>
struct case_< Image_value_kind, Exact, 3 >
  : where_< value::is_label<stc_get_type(value)> > {
  typedef LabelImage<Exact> ret;
};
template <typename Exact>
struct case_< Image_value_kind, Exact, 4 >
  : where_< value::is_color<stc_get_type(value)> > {
  typedef ColorImage<Exact> ret;
};
template <typename Exact>
struct default_case_< Image_value_kind, Exact > {
  typedef DataImage<Exact> ret;
};

// Selector #2: image dimension.
typedef selector<Image, 2> Image_dimension;

template <typename Exact>
struct case_< Image_dimension, Exact, 1 >
  : where_< stc_is_a(grid, Grid_1D) > {
  typedef Image1D<Exact> ret;
};
template <typename Exact>
struct case_< Image_dimension, Exact, 2 >
  : where_< stc_is_a(grid, Grid_2D) > {
  typedef Image2D<Exact> ret;
};
template <typename Exact>
struct case_< Image_dimension, Exact, 3 >
  : where_< stc_is_a(grid, Grid_3D) > {
  typedef Image3D<Exact> ret;
};

// Other selectors.
// ...
```

Fig. 7. A selector from the Olena library (C++ template metaprograms). `stc_is_a` relies on a Metalic algorithm to test the is-a relationship (see also Algorithm 1).

Delegation SCOOP 2 features a delegation mechanism allowing any concrete instance of a static hierarchy to declare another object as being its *delegatee* (the former object becoming then a *delegator* to the latter). For practical purposes, the delegator holds a reference to its delegatee.

The virtual type `delegatee`, when defined for a type `T` (e.g., set to type `D`), alters the behavior of STATIC. In fact, `T` may use information from zero, one or two of these branches:

- the inheritance branch, possibly defined by the `super` relationship ;
- the delegation branch, defined by the `delegatee` virtual type.

Thus, in addition to the virtual types of its (possible) super class, `T` inherits from the virtual types of `D` by default. In case of conflict (i.e., a virtual type defined in both branches), the inheritance supersedes the definition of the delegatee. Moreover, if `T` and `D` are linked to their abstractions thanks to STATIC's implicit link mechanism (see 3.3), `T`'s abstractions can be selected according to `D`'s properties, if the category they belong to defines rules based on virtual types defined by `D` and retrieved by `T`. Consequently, the paradigm can let `T` collect all or some of `D`'s abstractions, or even model concepts computed from `D`'s properties. Hence, the delegation mechanism allows a type to acquire *interfaces*, using the properties of the delegatee.

SCOOP 2 also allows a class to retrieve the *implementation* corresponding to these interfaces. The guidelines of a SCOOP 2-based library require ARCHITECTS of the library to have each abstraction `A` derive from the special type `automatic::get_impl<A, Exact>` (`Exact` being the exact concrete type of the class), provided by STATIC. By specializing another STATIC template, `automatic::set_impl`, ARCHITECTS can provide implementations for each method of the abstraction `A`, e.g. passing on the call to the delegatee. This way, SCOOP 2 allows library DESIGNERS to fully write real type transformations, not relying on direct inheritance, but on the properties of the transformed type. Instances of such implementation classes performing transformations are called *morphers* in the paradigm, while non-morpher implementation classes are called *basic types*. Morphers offer a new vision of the generic programming paradigm called *semantics-driven genericity*. These generic, composable and lightweight objects built on one or several images, are useful in a wide range of situations. For example, they can be used to implement:

mixins A morpher can add extra data (e.g. a neighborhood) or operations (e.g., an ordering on the values) to an image;

adapters E.g., a slice morpher can be used to view a slice of a 3-D image (spacemap) as a 2-D image (bitmap);

modifiers a morpher can add a mask to an image, to restrict its (iterable) domain;

lazy function applications A morpher can present an image seen through a function, either bijective or not;

More generally, morphers can be used to build three kinds of services.

extrusive equipment Changing the behavior of a generic algorithm non intrusively is a feature often wanted. Such a variation can be a logging facility tracing the execution of an algorithm, or an attached Graphical User Interface (GUI) display depicting the evolution of an image processing. An intrusive approach implies adding extra code in existing algorithms to handle the variations or maintain several versions of these algorithms, which is error-prone and tedious. Morphers provide an elegant, non intrusive solution: instead of altering the algorithms, one can decorate input data with the needed equipment, using a morpher. When the algorithms manipulate this new type, they trigger the expected additional actions (logging, updating a GUI, etc.).

lightweight replacements These are data types constructed over existing ones, saving the creation and copy of temporary variables, resulting in a gain of time and space. For instance, applying a generic algorithm only to the red channel of a color red-green-blue (RGB) image usually requires the creation and manipulation of an object duplicating the data of the red channel, and storing back the result of the algorithm in the original RGB image. Instead, one can use a morpher to create a lightweight (read-write) *view* on the RGB image presenting only the red channel information. The algorithm can then be applied seamlessly, as this new type presents an interfaces similar to the morphed image. This "adaptation" introduces very little, or even no run time cost, thanks to compiler optimizations such as inlining; and no duplication of initial data. Such types are named *replacements* as they do not appear as new data types to practitioners.

actual data (new types) As opposed to the previous item, morphers can be used to implement new types, built on existing types. An image region is such a type: it behaves as an images, and yet it requires an image object to be defined. Combined with the topological definition of a region, one can write an image region morpher from an existing image.

Figure 8 shows an example of a morpher. Beforehand we must equip the abstraction `Image` so as to retrieve implementations if needed (which is the case when the implementation is a morpher class).

```
template <typename Exact>
struct Image : public virtual Any<Exact>,
               public automatic::get_impl<Image, Exact>
```

Then we shall give the implementation in question, which is just a set of delegations to the morphed class in the case of identity-based morphers (i.e., whose default behavior is to copy the interfaces and implementations).

```
// Morpher kind tag.
namespace behavior { struct identity; }
// Automatically-retrieved implementation.
namespace automatic {
  template <typename Exact>
  struct set_impl<Image, behavior::identity, Exact>
    : public virtual Any<Exact>
```

```
  {
    stc_typename(psite);
    stc_typename(value);
    // Delegate the call.
    value impl_read(const psite& p)
    {
      return exact(this)->image()(p);
    }
    // ...
  };
}
```

Last, we can write the morpher, and use it like any other basic type.

```
image2d<bool> ima1;
casted_image< image2d<bool>, int > ima2 (ima1);
// ima2 has the interface of Image2D (among others).
bool b = ima2 (point2d (42, 51));
unsigned size = bbox_size (ima2);
```

Stopping the delegation When a virtual type, either defined in `T` or inherited from its super class, is set to **Not_Delegated**, the delegation branch is ignored for the resolution of this virtual type. This feature is useful from the software engineering point of view, when subclassing a base class for a kind of morpher, to force PROVIDERS of algorithms to define the virtual types that should be retrieved automatically from the delegator semantically.

For example, consider a basic library type class `image2d<Value>`, used to store a 2-dimensional image (with actual data in memory); and an abstract morpher template class `value_morpher<Image>`, having a delegatee of type `Image`. Types from the category `image` must define a virtual type `value`, which is perfectly defined for `image2d<Value>`, and is `Value`. However, this virtual type is set to **Not_Delegated** for `value_morpher<Image>`. Indeed, as this class serves to factor the development of morphers altering the values of an image (also called *value-wise transformations*), its subclasses shall define a valid virtual type `value`, and not use the value type of the morphed type (i.e., the delegatee) automatically. Now, let us imagine that a PROVIDER wants to implement a concrete `casted_image<Image, TargetValue>` morpher, whose purpose is to present an image of type `Image` as a (read-only) image having values of type `TargetValue`. Thus, `casted_image<image2d<int>, float>` would be a lazy transformation of an image of `int`s to an image of `float`s (no data is allocated, nor modified: an instance of this class is just a function of its delegatee `image2d<int>`). If this PROVIDER forgets to give the virtual type `value` of `casted_image<Image, NewValue>` a valid definition (presumably, `NewValue`), STATIC's equipment will trigger an error at compile-time.

3.4 Algorithms

Following the classical generic programming (GP) paradigm, SCOOP 2 expresses algorithms as function templates, not as methods. However, methods are used

```
// value_morpher.
// --------------
// Base class for morphers altering values.
template <typename Exact> class value_morpher;

template <typename Exact>
struct super_trait_< value_morpher<Exact> > { typedef image_base<Exact> ret; };

template <typename Exact>
struct vtypes< value_morpher<Exact> > {
  // This behavior makes subclasses of value_morpher
  // retrieve methods from their delegatee by default.
  typedef stc::final< behavior::identity > behavior;
  // This virtual type must be defined in subclasses.
  typedef stc::not_delegated               value;
};

template <typename Exact>
class value_morpher : public image_base<Exact> {
  typedef image_base<Exact> super;
public:
  stc_typename(delegatee);
protected:
  value_morpher() {};
};


// casted_image.
// -------------
template<typename I, typename U> class casted_image;

// casted_image gets properties from value_morpher...
template<typename I, typename U>
struct super_trait_< casted_image<I, U> >
{ typedef value_morpher< casted_image<I, U> > ret; };

template<typename I, typename U>
struct vtypes< casted_image<I, U> > {
  //...and from its delegatee.
  typedef I                  delegatee;
  // Redefine virtual types.
  typedef typename I::point point;
  typedef U                  value;
};

template<typename I, typename U>
class casted_image
  : public value_morpher< casted_image<I, U> > {
public:
  typedef casted_image<I, U> self;
  typedef value_morpher<self> super;
  stc_using(point);
  stc_using(value);
  stc_using(delegatee);
  casted_image(Image<I>& image)
    : image_ (*exact(&image)) {}
  value impl_read(const point& p) const {
    // Casted to the new value type (U).
    return image_(p);
  }
  protected:
    I& image_;
};
```

Fig. 8. An example of morpher.

for non-generic routines attached to objects and to access to internal data of objects. Generic algorithms benefit from the same traits of Scoop 1: abstraction-based constraints on inputs, covariant arguments, polymorphic associated types (`typedefs`), statically-dispatched multimethods. We do not discuss much of the subject because this paper focuses on the data structure aspect, and most of its contents is covered in [7]. Figure 9 gives a small example of abstraction-based overloading.

```
// Compute the number of points of an image.
template <typename I>
unsigned npoints (const Image<I>& input) {
  // Slow version iterating over all the points.
  // ...
}
// Specialized version for Image2D.
template <typename I>
unsigned npoints (const Image2D<I>& input) {
  // Fast version computing the result directly.
  return input.nrows () * input.ncols ();
}
```

Fig. 9. An example of abstraction-based overloading.

4 Introducing Static

This section presents Static, a component to build libraries using the Scoop 2 paradigm. For space reasons, we try to provide as much insight as possible, but the article cannot cover all the details of the implementation. For instance, some metaprogramming algorithms are given in a synthetic form, in place of the longer and complex original C++ template code[1].

4.1 Equipment

To use static within a library, one must *equip* a `namespace` with some types and functions. This prevent the mechanism from polluting the global namespace. In this namespace, the library Designer can declare new virtual types to be used as properties of Scoop 2 classes.

```
// Macros.
#include <stc/scoop.hh>
namespace my {
  // Equip with types and functions.
  #include <stc/scoop.hxx>
  // Declare the virtual types used in this context.
```

[1] Interested readers may want to consult the original code at http://trac.lrde.org/olena/wiki/Static.

```
  mlc_decl_typedef(grid);
  // ...
}
```

`mlc_decl_typedef` uses a technique similar to the typedef introspection technique from [28].

4.2 Static hierarchies

Static provides a class `Any` to make the construction and use of static hierarchies easier. The top-most classes of the hierarchies of the library shall inherit from `Any`, so that the exact type is available through the `exact` function.

```
template <typename Exact> struct Any {};
template <typename Exact> Exact* exact(Any<Exact>* ref)
{ return (Exact*)(void*)ref; }
```

This is admittedly the most intrusive point in using Static within a library, as the C++ language does not allow to add super classes a posteriori. Thus, the equipment of existing libraries might be tedious and require wrapping the existing classes.

4.3 Virtual types

Virtual types for the type `T` are defined as a specialization of the class `vtypes<T>`, defined in the `stc/scoop.hxx` equipment.

```
struct point2d;
template<> struct vtypes<point2d> {
  typedef mlc::uint_<2> dim;
  typedef int           coord;
  typedef grid2d        grid;
};
```

Semantically, a class inherits from the virtual types of its super class, but this inheritance is implicit: the user needs not mention it, since this task is taken care of by Static's virtual type look-up algorithm (see Algorithm 2). To make the recursive retrieval of virtual types possible, Static cannot simply rely on the C++ inheritance relationship. The language doesn't provide any means to actually retrieve the type of a super class. Hence, Static needs the author to inform a special traits to keep this information, `super_traits_`. This relationship shall not mirror the C++ inheritance exactly: it shall be limited to *implementation classes*, as they are the only classes *defining* properties. A class having no implementation super class must inherit from the special type **None** (`mlc::none`).

```
template<>
struct super_trait_<point2d> { typedef mlc::none ret; };
```

Virtual types can take any "value" (i.e. C++ type, defined by the language or the user). However, STATIC uses some special values (the C++ names are given in parenthesis).

Abstract (`stc::abstract`) Used to declare an abstract virtual type.

Final(*val*) (`stc::final<val>`) **Final** is a qualifier that does not change the value of the virtual type defined, but prevents any subclass or delegator to redefine the virtual type.

Not_Delegated (`stc::not_delegated`) Prevents the lookup algorithm from using the delegation branch to retrieve a virtual type; only the inheritance branch will be used.

Virtual types are domain-related; their meanings have no impact on STATIC, except **delegatee** (`delegatee`). This special virtual type is used to attach a *delegation branch* to a given class and its subclasses (see 3.3). The retrieving of the virtual type `T` from class `C` uses the lookup algorithm FIND. If no error occurred during the lookup (e.g., due to a erroneously-designed hierarchy) FIND(`C`, `T`) returns the "value" of the virtual type if found, **Not_Found** (`mlc::not_found`) otherwise. Algorithm 2 shows FIND as a pseudo-algorithm. The translation to the corresponding C++ metaprogram is immediate.

FIND makes use of several routines and values. The ones that are not defined in Algorithm 2 are described hereinafter.

Super(*type*) (`super_traits_<type>::ret`) This function returns the super class of *type*.

None (`mlc::none`) The value returned by a class having no super class (in the STATIC acceptation).

Find(*source*, *target*) (`find<source, target>::ret`) Recursively look for the value of the virtual type *target* for class *source*. The virtual type is searched in the inheritance branch as well as in the delegation branch (if it exists). FIND expects a virtual type to have a concrete definition. If the virtual type lookup ends up with **Abstract**, FIND triggers a compile-time error.

Find-Local(*source*, *target*)
(`find_local<source, target>::ret`) Query the class *source* for the value of the virtual type *target* directly, using `vtypes`. This algorithm is used by FIND.

Not_Found(`mlc::not_found` The value returned by FIND when a virtual type is not found.

In addition to FIND, STATIC proposes a CHECK algorithm performing additional checks not directly required by the lookup task (see Algorithm 3).

Macros As STATIC is almost only composed of template types, we wrote several macros to serve as syntactic sugar. This section explains the meaning of each macro used in the paper.

`stc_typename` and `stc_using` are just shortcuts to equip classes: they respectively inject a virtual type in the scope of the current class and retrieve a virtual type from a super class.

```
FIND(source, target)
 1  if source = delegatee
 2    then return SUPERIOR-FIND(source, target)
 3  (where, res) ← FIRST-STM(source, target)
 4  switch
 5    case res = Not_Found :
 6          return DELEGATOR-FIND(source, target)
 7    case res = Abstract :
 8          res_delegator ← DELEGATOR-FIND(source, target)
 9          if res_delegator = Not_Found
10            then error "target is abstract."
11    case res = Not_Delegated :
12          return SUPERIOR-FIND(source, target)
13    case default : return res

DELEGATOR-FIND(source, target)
 1  deleg ← SUPERIOR-FIND(source, delegatee)
 2  if SUPERIOR-FIND(source, delegatee) = Not_Found
 3    then return Not_Found
 4    else  return FIND(deleg, target)

SUPERIOR-FIND(source, target, current = source)
 1  if current = None
 2    then return Not_Found
 3  stm ← FIND-LOCAL(current, target)
 4  switch
 5    case stm = Abstract :
 6          error "target is abstract"
 7    case stm = Not_Found or stm = Not_Delegated :
 8          sup ← SUPER(current)
 9          return SUPERIOR-FIND(source, target, sup)
10    case stm = Final(val) :
11          return val
12    case default : return stm

FIRST-STM(source, target)
 1  if source = None
 2    then return (None, Not_Found)
 3  stm ← FIND-LOCAL(source, target)
 4  switch
 5    case stm = Not_Found :
 6          return FIRST-STM(SUPER(source), target)
 7    case stm = Final(val) :
 8          return (source, val)
 9    case default : return (source, stm)
```

FIND-LOCAL(*source*, *target*) locally queries the *source* class for the virtual type *target*. If the set of virtual types attached to *source* (i.e., `vtypes<`*source*`>`) has a definition for *target*, this "value" is returned, otherwise **Not_Found** is returned.

Algorithm 2: Virtual type lookup.

```
CHECK(source, target)
 1  if source = None
 2    then return      — Stop condition.
 3  stm ← FIND-LOCAL(source, target)
 4  sup ← SUPER(source)
 5  switch
 6    case stm = Abstract :
 7         orig ← (source, Abstract)
 8         CHECK-NO-STM-DEFINED(orig, sup, target)
 9    case stm = Final(val) :
10         CHECK-FINAL-STM(val)
11         orig ← (source, val)
12         CHECK-NO-FINAL-DEFINED(orig, sup, target)
13    case stm = Not_Delegated :
14         orig ← (source, Not_Delegated)
15         CHECK-NO-FINAL-DEFINED(orig, sup, target)
16    case stm = Not_Found :
17           — Nothing.
18    case default :
19         orig ← (source, stm)
20         CHECK-NO-FINAL-DEFINED(orig, sup, target)
21
22  return CHECK(sup, target)

CHECK-NO-STM-DEFINED(orig, source, target)
 1  if source = None
 2    then return      — Stop condition.
 3  stm ← LOCAL-FIND(source, target)
 4  if stm ← Not_Found
 5    then error "target re-declared abstract in orig."
 6    else  sup ← SUPER(source)
 7          CHECK-NO-STM-DEFINED(orig, sup, target)

CHECK-FINAL-STM(source, target, stm)
 1  if stm = Abstract
 2    or stm = Final(val)
 3    or stm = Not_Delegated
 4    or stm = Not_Found
 5    then error "Ill-formed final vtype."

CHECK-NO-FINAL-DEFINED(orig, source, target)
 1  if source = None
 2    then return      — Stop condition.
 3  stm ← LOCAL-FIND(source, target)
 4  if stm = Final(val)
 5    then error "Final vtype target redefined in orig."
 6    else  CHECK-NO-FINAL-DEFINED(orig, SUPER(source), target)
```

Algorithm 3: Additional type-checking rules.

```
# define stc_typename(T) typedef stc_type(Exact, T) T
# define stc_using(T)    typedef typename super::T  T
```

`stc_type (source, target)` and `stc_deferred (source, target)` are macros performing the same task: they call the template metaprogramming algorithm FIND and expand as the result of the lookup. However, the former also calls the CHECK algorithm (Algorithm 3) to ensure the rules given above are followed. Because of the recursive nature of the approach, we cannot always perform all checks on virtual types (C++ compilers consider types coming across their own definition through successive type instantiations as *empty*, which would break our algorithms). Hence we use `stc_deferred` in problematic cases (virtual type definitions, for instance).

4.4 Extensible inheritance

SCOOP 2 features an extensible inheritance mechanism used to link the top classes of implementation hierarchies to the right abstractions. The entry point of this system is the class `top<Exact>`, which recursively inherits from the results of the static `switch` statements (selectors) tagged `internal::selector<Exact,` n`>`, where n covers the first values of $\mathbb{N}^*$. The technique used is similar to generating scattered hierarchies (`GenScatterHierachy`) exposed in [23].

5 Related Work

Many generic programming techniques have been invented to develop C++ libraries. First of all, traits [29] have been widely used to attach properties and associated types to generic types, notably within the Standard Template Library [2].

The idea of adding and checking constraints on the parameters of C++ templates with respect to a given contract (structural conformance, name conformance) is not new, and has led the way to the current work on future C++0x concepts [13]. McNamara and Smaragdakis have proposed a solution based on static interfaces [30]. Siek and Lumsdaine have formalized the principle of concept checking [16] within the Boost Concept Check Library (BCCL). Building on similar metaprogramming techniques, a new form of concept-based polymorphism can be obtained [17]. One of the advantages of these solutions is that they require no language extension, and can therefore be applied under the form of portable libraries with C++ compilers conforming to the 2003 ISO/IEC standard. Many of them will be superseded by the use of reified concept features in the forthcoming C++ standard.

As a matter of fact, several of these techniques are used within the OLENA project: traits serve to define properties (although querying their value requires a non-trivial metaprogramming algorithm); Static interfaces and concept-checking are part of SCOOP 2 thanks to the Generalized Curiously Recurring Template Pattern.

Static introspection is a technique implemented in Metalic and used by Static to retrieve properties. Zólyomi and Porkoláb have described a framework providing these services (among others) as part of a C++ library [28].

Last, the extensible inheritance in Static is based on a technique similar to the typelists described by Alexandrescu in [23] and which has proved to be useful to design recursively-defined class hierarchies [31].

Several existing libraries have proposed solutions to express type transformations [2,32,6]. However, all these approaches create types whose interfaces are limited to a known abstraction, and do not handle the specificity of the original type if it conformed to a subabstraction.

6 Conclusion

We have presented an extended generic programming paradigm (Scoop 2), allowing library designers to express generic type transformations called morphers. The paradigm relies on the presence of semantically-enhanced data types thanks to properties expressed as virtual types in implementation classes. These properties are used to connect concrete classes to the abstractions they model. As abstractions are expressed as actual C++ classes, they can provide implementations as well, based on the nature of the object, thanks to a delegation mechanism which is part of the paradigm.

Scoop 2 is intended for designers of new libraries; it might not be easy to adapt to existing libraries, though the task can be achieved by wrapping existing classes. Indeed, we have successfully applied this approach on a small subset of the standard C++ library.

We provide a software component to help users design their libraries using Scoop 2 and reify idioms of the paradigm, Static. This component has been conceived in the context of the Olena project. We are working on providing syntactic sugar for the constructs of the paradigm using a language masking the verbosity of the C++ templates; however, we don't want to make this extra language mandatory, and we will keep Static as a pure C++ project, since library-based language extensions are generally less costly and more sustainable than domain-specific languages [33].

We ran our tests using the GNU C++ Compiler (GCC) version 4.1 on Debian GNU/Linux, and we are working on porting Static to other configurations, notably using the Intel C++ Compiler (ICC). The paradigm itself is expressed solely using C++03 constructs. We have conducted some experiments with the ConceptGCC compiler [34] to check whether Scoop 2 can benefit from C++0x new features, but the compiler did not properly support mixing inheritance and where-clauses at that time.

Acknowledgments

We thank Alexandre Duret-Lutz, Olivier Gaça, Maxime van Noppen, Benoît Sigoure and Didier Verna for their proofreading and comments.

References

1. ISO/IEC: ISO/IEC 14882:2003 (e). Programming languages — C++ (2003)
2. Stepanov, A., Lee, M., Musser, D.: The C++ Standard Template Library. Prentice-Hall (2000)
3. Project, T.B.: Boost C++ libraries. http://www.boost.org/ (2008)
4. Siek, J.G., Lumsdaine, A.: The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In: International Symposium on Computing in Object-Oriented Parallel Environments. Number 1505 in Lecture Notes in Computer Science (1998) 59–70
5. The CGAL Project: CGAL, Computational Geometry Algorithms Library (2008) http://www.cgal.org.
6. Abrahams, D., Siek, J.G.: Policy adaptors and the Boost Iterator Adaptor Library. In: Second Workshop on C++ Template Programming. (October 2001)
7. Burrus, N., Duret-Lutz, A., Géraud, Th., Lesage, D., Poss, R.: A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming. In: Proceedings of the Workshop on Multiple Paradigm with Object-Oriented Languages (MPOOL), Anaheim, CA, USA (October 2003)
8. Duret-Lutz, A.: Olena: a component-based platform for image processing, mixing generic, generative and OO programming. In: Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE)—Young Researchers Workshop; published in "Net.ObjectDays2000", Erfurt, Germany (October 2000) 653–659
9. Géraud, Th.: Advanced static object-oriented programming features: A sequel to SCOOP. http://www.lrde.epita.fr/people/theo/pub/olena/olena-06-jan.pdf (January 2006)
10. Lombardy, S., Régis-Gianas, Y., Sakarovitch, J.: Introducing Vaucanson. Theoretical Computer Science **328** (November 2004) 77–96
11. Claveirole, Th., Lombardy, S., O'Connor, S., Pouchet, L.N., Sakarovitch, J.: Inside Vaucanson. In Springer-Verlag, ed.: Proceedings of Implementation and Application of Automata, 10th International Conference (CIAA). Volume 3845 of Lecture Notes in Computer Science Series., Sophia Antipolis, France (June 2005) 117–128
12. Siek, J.G., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. 1st edn. C++ In-Depth Series. Addison Wesley Professional (December 2001)
13. Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Reis, G.D., Lumsdaine, A.: Concepts: Linguistic support for generic programming in C++. In: Proceedings of the 2006 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press (October 2006) 291–310
14. Garcia, R., Järvi, J., Lumsdaine, A., Siek, J.G., Willcock, J.: A comparative study of language support for generic programming. In: Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, And Applications (OOPSLA), New York, NY, USA, ACM Press (2003) 115–134
15. Järvi, J., Gregor, D., Willcock, J., Lumsdaine, A., Siek, J.: Algorithm specialization in generic programming: challenges of constrained generics in C++. In: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Ottawa, Ontario, Canada, ACM Press (June 2006) 272–282

16. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: Proceedings of the First Workshop on C++ Template Programming, Erfurt, Germany (October 2000)
17. Järvi, J., Willcock, J., Lumsdaine, A.: Concept-controlled polymorphism. In Pfennig, F., Smaragdakis, Y., eds.: Generative Programming and Component Engineering (GPCE). Volume 2830 of LNCS., Erfurt, Germany, Springer-Verlag (September 2003) 228–244
18. Coplien, J. In: A Curiously Recurring Template Pattern. In [35].
19. Chiba, S.: A metaobject protocol for C++. In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). SIGPLAN Notices 30(10), Austin, Texas, USA (October 1995) 285–299
20. Veldhuizen, T.L.: Techniques for scientific C++. Technical Report 542, Indiana University Department of Computer Science (August 1999)
21. EPITA Research and Developpement Laboratory (LRDE): The Olena image processing library. http://olena.lrde.epita.fr (2003)
22. EPITA Research and Developpement Laboratory (LRDE): A prototype using SCOOP 2 and the C++ standard library. https://trac.lrde.org/olena/wiki/SCOOP/MiniStd (2007)
23. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)
24. Gurtovoy, A., Abrahams, D.: The Boost MPL library. http://www.boost.org/libs/mpl/doc/index.html (2004)
25. David Abrahams, A.G.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. 1st edn. C++ In-Depth Series. Addison Wesley Professional (December 2004)
26. Adobe: The Adobe Generic Image Library (GIL). http://opensource.adobe.com/gil/ (2007)
27. Siek, J.G., Lumsdaine, A.: Modular generics. In: Concepts: a Linguistic Foundation of Generic Programming, Adobe Systems (April 2004)
28. Zólyomi, I., Porkoláb, Z.: Towards a general template introspection library. In Karsai, G., Visser, E., eds.: Generative Programming and Component Engineering (GPCE). Volume 3286 of LNCS., Vancouver, Canada, Springer-Verlag (October 2004) 266–282
29. Myers, N.C.: Traits: a new and useful template technique. C++ Report **7**(5) (June 1995) 32–35
30. McNamara, B., Smaragdakis, Y.: Static interfaces in C++. In: First Workshop on C++ Template Programming, Erfurt, Germany. (October 10 2000)
31. Zólyomi, I., Porkoláb, Z., Kozsik, T.: An extension to the subtype relationship in C++ implemented with template metaprogramming. In Pfenning, F., Smaragdakis, Y., eds.: Generative Programming and Component Engineering (GPCE). Volume 2830 of LNCS., Erfurt, Germany, Springer-Verlag (September 2003) 209–227
32. Weiser, M., Powell, G.: The View Template Library. In: First Workshop on C++ Template Programming, Erfurt, Germany (October 2000)
33. Stroustrup, B.: A rationale for semantically enhanced library languages. In: Proceedings of the Workshop on Library-Centric Software Design (LCSD), San Diego, California, USA (October 2005)
34. Indiana University: ConceptGCC. http://www.generic-programming.org/software/ConceptGCC/ (2007)
35. Lippman, S.B., ed.: C++ Gems. Cambridge Press University & Sigs Books (1998)