# Multistage Algorithms in C++

Dissertation
zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultäten
der Georg-August-Universität zu Göttingen

vorgelegt von
Andreas Priesnitz
aus Göttingen

Göttingen 2005

D7

|  |  |
|---|---|
| Referent: | Prof. Dr. G. Lube |
| Korreferent: | Prof. Dr. R. Schaback |
| Tag der mündlichen Prüfung: | 2. November 2005 |

*Andreas Priesnitz*

# Multistage Algorithms in C++

*in commemoration*
*of my father,*
*Kuno Priesnitz*

# Contents

# Chapter 1

# Frequently Asked Questions

The format of the preface to this text is a tribute to the author's liking of structured and distributed documents, in favour of abstracting content from (offline or online) representation.

## 1.1 Who is it for?

To allow the reader to verify whether this text suits her or his needs and abilities, a brief classification of its applications and prerequisites is given here, which should not reject the generally interested, though.

### 1.1.1 Domain of interest

Basically, this text presents a new approach to expressing program code. It is not a new programming language, as it is formed by a strict subset of ANSI C++, neither is it a library of particular application domain. It rather is a collection of cooperating constructs that are intended to serve as a partial layer on top of the C++ language. As such, it serves on the one hand to encapsulate C++ elements and powerful programming techniques which are tedious to use, on the other hand it allows to write code at a higher degree of expressiveness. Hence, it may be considered a new convention of "speaking" the C++ language in order to say more in less words. A motivation of why one might want to use such an approach at all is given in § 1.2.

This effect may, naturally, be beneficial to software implementors, given that their interests are covered by that layer—which is just partial, as mentioned above. Generally, it extends to *statically bound applications*, i.e., whose participating elements are designated at compile time:

**Performance-critical software** has to be implemented avoiding dynamic binding, at least in its frequently evaluated parts, to avoid run-time and memory overhead and to allow optimizations, as will be illustrated in § 2.2.3. There is a wide range of such applications, from operating systems to games, from compilers to numerical simulations, from data bases to image processing. A very typical selection constitutes the SPEC CPU2000 benchmark [SPE].

**Libraries** (should) provide well-designed data structures and algorithms that (should) cover a large class of applications. This usually implies the supply of alternative implementations to optimally support different cases of application, which have to be realized at least to a certain extent by statically bound code that explicitly refers to the particular representation.

The implication of the ideas presented in this text is highest, of course, if one is concerned with the intersection of both areas, i.e., with the creation of well-performing, widely applicable software.

### 1.1.2 Domain of knowledge

Given that C++ serves as base to the approach to be presented and given that C++ is far too complex to be introduced here, the reader is expected to have a certain familiarity with that language. While many of its aspects will not be considered at all and many of the considered aspects are not proprietary to C++, selected topics may exceed the understanding of someone just aquaint with some other object-oriented programming language or with C. In any case, one should know peculiarities of C++ like:

- references vs. pointers
- multiple inheritance
- virtual functions
- typedefs
- access restriction
- friend functions
- preprocessor macros
- namespaces
- operator overloading

Besides the plain programming language skills, one probably needs to have some experience in developing software for the domains specified in § 1.1.1 and with the inherent problems of doing so, in order to bear the profound will to follow rather technical discussions to learn about according solutions.

## 1.2 What is it for?

Different people have very different perceptions of the term *software quality*, according to the way they are concerned. Usually, their common and complete satisfaction is impossible, as characterizing features may be opposed, requiring to assign them an application-dependent priority. While this is a general subject of software engineering, a very prominent conflict shall be in focus of this discussion. Topics like portability, error handling or documentation are not explicitly addressed, while the methods to be presented will be developed at least under the requirement of not violating common rules of handling them.

The following explanations may actually go beyond the scope of those methods, but are provided to clearify common misunderstandings encountered in discourses.

### 1.2.1 Efficiency

*End users* are barely concerned with source code design. Usually, just qualities of the executable code matter to them. In the context of performance-critical software, this is dominated by the request for high *execution efficiency*, in particular for the minimization of execution time and memory consumption. These two qualities may need a trade-off:

- In many cases, faster algorithms involve more storage of auxiliary data than slower alternatives.

- If some constant derivable data is requested frequently from a data structure, this information might be created once and stored explicitly in the structure, so that it will just be read rather than derived on each access, which might improve execution speed and surely costs memory space.

But they may go hand in hand as well:

- Large or distributed data structures require more frequent loads of data into higher levels of the memory hierarchy (registers or cache) or a higher band-width of transmissions (from external storage devices or through a network), which may result more expensive than to derive the data from other data that is more easily available.

- Similarly, executable code has to be loaded like data and might perform worse if being too large (e.g., by excessive *loop unrolling* or *inlining*).

- A problem of given size might need much more complex (and expensive) algorithms in order to be solved in parts if memory is exhausted due to the too large (featureful) choice of data structures.

With respect to the *application programmer*, important performance issues are time and memory consumption of the compilation process, considered as *compilation efficiency*. While they depend strongly on the compiler used, some programming manners have immanent consequences for these costs. When deliberating upon minimizing them, though, they have to be set in relation to the additional costs of human work to obtain equivalent code.

Frequently, the techniques that will be presented in the oncoming are criticized for their high impact on compilation costs, neglecting the time equivalent hand-crafted optimizations would take. Furthermore, the compiler's performance is accelerated by Moore's law, while a programmer's speed remains more or less constant, using a particular language.

The *manager* has to calculate costs of human working time as well as the economical interest in minimizing both the delay until the software exists and further maintenance obligations. The creation of code requires much more effort if it has to be very efficient than if its perfomance is of minor relevance. Usually, it is more effective to invest the developer's time in optimizing structural and algorithmic decisions rather than in arranging low-level implementation details. Thus, the latter burden should be taken off her shoulders, which can be put as a demand for *programming efficiency*, leading to a different class of primary features to consider.

### 1.2.2 Abstraction

From the *developer's* point of view, software quality refers to design and properties of the source code which are hard to measure by an absolute scale:

***Modularity/Locality*:** How large is the scope of individual implementation details? It is desirable that modifications (e.g., corrections) require only local changes, which is particularly essential when the modifying person will not be the one who developed the according parts of the software. This quality is supported by the *object-oriented programming* paradigm.

***Scalability / Extensibility*:** How much work does it cost to widen the domain of the software to other types of data or methods? Usually, software should be designed to allow for extensions. Though being related to the problem of modularity, this rather refers to the software's semantics; even very well modularized code might need to be replicated in large parts to be able to deal with slightly different tasks. This quality aims at future developments and is therefore particularly essential for long-term projects.

***Reusability / Minimality*:** How often can parts of the software be reused in different contexts, and how easy is it to do so? The more general the design of algorithms and data structures, the wider is their applicability. Ideally, new software can be implemented by combining existing parts and adding as few adaptations as possible (depending on the scalability). Conversely, this goal can be viewed as the minimization of code similarities. It is achieved by *abstracting* the problem and providing an as general implementation as possible, which usually results in referring to formalized models and embedding eventual specializations.

***Expressiveness*:** This term describes the "power" of the provided constructs, not by their semantics, but in terms of the condensedness of representation. The less separate steps have to be combined to achieve a particular effect, the better.

***Complexity*:** How hard is it to apply the software? The optimal choices of data structures and algorithms depend on each other and require knowledge which may not necessarily be expected of the developer if she is not the one who provided their implementations. It is desirable to automatize this decision process to make it possible to compose data structures and algorithms just by their semantical (implementation-independent) properties.

***Clarity*:** How difficult is it to understand the software? This does not address the documentation, which definitely is a primary issue but not directly related to software design. Well-structured software is far more easy to understand than such with many arbitrary constructs.

***Stability*:** Finding (semantical) errors and flaws is heavily supported by a strict design that avoids ambiguities or unexpected states beforehand. This includes, but is not limited to debugging support and error handling, which are particular technical matters that do (or should) not influence the general programming methodology.

### 1.2.3 Situation

The quickly increasing size and complexity of software projects demands for more thorough consideration of their design. Since advanced design implies higher levels of abstraction in the description of data structures and algorithms, its realization traditionally conflicts with the requirement of high performance, as it usually is translated to indirections causing overhead. For a long time, programming languages did not provide means to solve this conflict in a satisfactory way, which led to the common opinion that it were not soluble at all.

Fortunately, recent observations and achievements promise that this is not true, if the *generic programming* paradigm is applied, which allows to freely abstract algorithms and data structures and to process static information to generate code whose efficiency is comparable to that of a hand-crafted version, if a well-optimizing compiler is employed. These advantages facilitate the development of widely applicable and highly efficient software.

Now, structure may even be beneficial to performance. Complex algorithms and data structures may be very hard and error-prone to implement and optimize by hand. If they can be composed from existing (efficient) building blocks without causing performance overhead and if optimizations can be encapsulated in reusable entities, this would not only save time for coding and debugging, but leave the responsibility of creating efficient and syntactically correct code to the compiler (who is an accredited expert on C++ syntax) and let the programmer concentrate on higher-level optimizations when composing the parts. This allows very complex optimizations, which theoretically could be implemented by hand as well, but which are too tedious to be realized within a given time span.

Consequent design of efficient generic software leads to general questions concerning the systematic embedding of according programming techniques. Various solutions were suggested in popular examples of generic software for small entities (numbers, smart pointers) or regular, shallow structures (containers, strings, vectors, matrices, tensors). When trying to implement more complex applications, though, new problems arise which indicate that those solutions are hardly scalable. Existing implementations rely on simplifying assumptions or external knowledge of encapsulated details. A common problem, answered differently in different languages, is that of *dispatching* the appropriate implementation of an algorithm out of a set of different alternatives, depending on the choice of data structures that may serve as its arguments.

### 1.2.4 Quest

A systematic approach is sought that does not only allow such a *multiple dispatch*, but that also provides the benefits of object-oriented and generic programming techniques in a general manner, to make them independent of the target application. This text gives an overview of different techniques and suggests a new programming strategy that combines their benefits in a both general and constructive way. It will have to be reflected in the design of both algorithms and data structures and requires to be consequently supported starting from the lowest level, which are the elements provided by the C++ language specification. As the limitations of other design concepts are to be overcome, their integration is to be considered secondary to a consequent solution.

While the problem of how to define data structures of appropriate format, though relevant and interesting, shall not be examined in this place, the programming method to present will aim at the creation of algorithms, which is done naturally by composing them of simpler ones. At the same time, expressiveness is a major goal, i.e., the resulting notation should be both intuitive and compact, saving the application programmer from learning overly complex technical details. This will be achieved by mechanisms that are performed at different *stages* of the compilation process, which justifies the term *multistage algorithm* for the subject of discussion.

## 1.3 Where is it from?

While this text describes a particular programming method which is implemented in a standalone manner, basing only on ANSI C++, its motivation and application rings many bells, some of which are indicated here.

### 1.3.1 Related subjects

This text mainly deals with topics in the fields of *software technology* and *programming languages*, or better, *programming paradigms*. Of the latter, *object-oriented*, *generic*, *meta*, *functional* and *literate programming* will be directly addressed, while the relation to *aspect-oriented programming* (*AOP*) is not a dependency, but an analogy. Both approaches belong to the idea of *post-object programming* (*POP*) and play a role in *component-based software engineering*. On the other hand, providing optimizations on language-level resembles methods known from *high-performance computing* and *compiler construction*.

Although the application domain is left open intentionally, it should be mentioned that the author's investigations were inspired by works in scientific computing motivated by problems of computational fluid dynamics. Their domain extended from numerical linear algebra methods for block matrix solving [Pri96, Kli01] over combinatorial graph methods for flow-based reduction [Sch00] to computational geometry methods for grid generation and refinement [Ber00, Koh00, Par02]. Particularly, the examination of software engineering questions like the generic composition of data structures [Pri01] and object-oriented distribution of data structures [Rei04] brought up many problems to be solved by this work.

### 1.3.2 Related information

Concerning C++, the current ANSI C++ language standard [ISO98] is indespensable to clearify many technical questions in detail, while [Str00] is the most popular and recommendable introduction to virtually all relevant aspects of the language. There is a considerable amount of qualitative online material on C++, including tutorials, a recommendable overview of which is given in [Moh]. In [Mey97, Mey95] and [Sut00, Sut02], advanced techniques and details are explained in a stand-alone manner.

With regard to the subjects of this text, publications on particular issues will be referenced within the according context, while general presentations on these topics and others are given in [Vel99, CE00, Ale01, Pri03, VJ03, AG05].

To keep up with actual developments in this area one should refer to the discussions in [ISO] on C++ and in [Boo] on techniques and applications.

### 1.3.3 Related software

Generic programming was popularized by the design of the *Standard Template Library* (*STL*) [SL95, Aus98, Jos99], which was soon adopted within the C++ standard. It offers a wide range of containers (e.g., vectors, lists, strings, sets), algorithms on sequences and functoids. This example initiated the creation of many other generic libraries for very different kinds of domain.

Many of them are hosted by the *Boost* project [Boo], which bears potential extensions to the STL from these. In particular, the *Boost Lambda Library* (*BLL*) [JP] is strongly related to the project developed within this text.

Non-Boost examples of generic libraries are listed in [Velb], only two of which shall be mentioned explicitly here, being the most prominent examples of generic libraries for linear algebra, the classic of performance-critical applications:

- *Blitz++* [Vela, Vel01] offers dense tensors of fixed or variable size.
- The *Matrix Template Library* (*MTL*) [Sie, SL98, Sie99] provides a wide range of matrix formats and vectors of variable size.

Both libraries' performance is comparable to that of hand-optimized C or Fortran libraries, while they provide far higher expressiveness by the use of expression templates. Although different successor projects like [WK, GLA] have been started meanwhile, their functionality has not yet been superseded significantly.

## 1.4 How is it done?

### 1.4.1 Structure

The document is split into three parts, of which the first summarizes common programming paradigms and typical problems in the sense of § 1.2 when applying them. It is given in a rather introductory style to allow the unfamiliar reader to get a grasp of those, as this should—by the author's experience—be helpful to the majority of the target group.

The second part introduces the proposed approach and its realization, forming the core of this method. Part three finally demonstrates how the techniques are applied to provide the elementary operations on which any further application may be based on.

### 1.4.2 Language

The programming techniques presented in this text are realized in C++. There is no real alternative to this choice, as they require the implementation language:

- to be object-oriented,
- to offer free (non-restricting) class and function templates of type and integer value parameters which are statically bound,
- to provide alternative constructs to express language concepts, both abstract ones involving overhead (virtual functions, dynamic memory allocation, recursion) and direct ones (plain functions, static memory allocation, loops),
- to allow to model low-level details and
- to be supported by some well optimizing compiler across many platforms.

Only C++ fulfills all of these requirements. But some negative aspects of C++ should be mentioned beforehand:

- It is a rather complicated language, due to its required compatibility to C, to many difficulties originating from its low-level facilities (e.g., pointers, references, hardware-dependent builtin types, ...) and to the fact that it provides alternative realizations of language concepts. It has to be understood that these are the price to pay for gaining the necessary direct influence on memory management and elementary instructions to be performed in order to provide highly efficient implementations, while the language also allows to introduce abstractions to encapsulate such details.

- The language is evolving (not at least to support developments as the ones presented here), and even if some feature has been standardized, the compiler vendors might not have it implemented yet in their products. Hence, it takes workarounds (which mess up the code) to have the code translated by a non-standard-compliant compiler.

  Fortunately, there are some compilers that support the actual standard very closely, like the *GNU Compiler Collection* (*GCC*), which is both free under the terms of the *GNU Public License* (*GPL*) and supported for many architectures and systems.

- It is still temptingly easy to leave the way of good program design, because C++ permits almost any C hack and the language takes less of one's burden of self-control as other languages do. In the area of high-performance-computing, though, being the dominion of Fortran and C, this should usually not be considered a major problem by the disciplined programmer.

This is, of course, the opinion of someone thinking in favour of C++. Much harsher critique of the language is found in [Joy99], but it mostly does not really apply to the subjects of this text.

### 1.4.3 Layout

Both the developed code and its documentation, which is the present text, are generated from the same source by means of the *literate programming* tool *noweb* [Ram, Ram94], which displays code enclosed in *chunks* that are tagged by titles enclosed in angle brackets ⟨...⟩. Chunks may be referenced by this tag, which causes the chunk to replace its reference when code is created (*tangling*). A ≡ attached to the tag name introduces a chunk definition, and a chunk's contents are extended by another definition marked by a +≡ symbol. Accordingly, the principal code file `cong/cong.hh` looks as follows.

```
⟨cong/cong.hh⟩≡
    #ifndef CONG_HH
    #define CONG_HH

    #ifdef HAVE_CONFIG_H
    #include <config.h>
    #endif

    ⟨⟨includes⟩⟩

    namespace cong
    {
        ⟨implementation⟩
        ⟨⟨undefine auxiliary macros⟩⟩
    }

    #endif
```

Double angle brackets ⟨⟨...⟩⟩ enclose code parts which are not discussed in this documentation for being trivial or analogous to other parts.

# Part I

# Paradigms

# Chapter 2

# Object-oriented programming

The *object-oriented programming* (*OOP*) paradigm is the natural way of answering the request for modular software design, excluding the numerous problems caused by universal accessibility of implementation details in *procedural programming*. An *object* is a self-governing unit of information which actively communicates with other objects, rather than being passively accessed as in a *procedural programming language*. The behaviour of an object is determined by assigning it a *type*, a formal description of its properties. The elementary building blocks of the type system are builtin types, which are implicitly defined by the language specification and which for C++ will be discussed in detail in § 6.1.1.

## 2.1 User-defined types

In contrast to a builtin type, a C++ *user-defined type* is explicitly defined, either as an enumeration type or as a class type.

### 2.1.1 Enumeration types

An *enumeration type*, though or just because being quite a simple type, is rather inconvenient to handle. On the one hand, it is represented by a builtin type internally and therefore does not allow to represent more complex information and to expose object-oriented behaviour. In most contexts, its instances behave by default like being of its *underlying type*, [ISO98] § 7.2;8. On the other hand, for being user-defined, some of its properties may or even have to be specified explicitly, wherefore it has to be given special attention in according constructs. A particular problem is the uniqueness of an enumeration type, [ISO98] § 7.2;4, which in contrast to class types prevents to express implicit relationships of such types to each other.

A significant advantage of an enumeration type outlined in [VJ03] § 17.2 is the fact that its values are true *rvalues*, in contrast to a "constant variable" declared to be non-modifiable by the *qualifier* `const` and represented as an addressable *lvalue* consuming space in memory. Therefore, it is the means of choice to represent (integral) constants—of non-builtin type, though.

Further implications of the mentioned duality will be commented where they apply. It is generally recommended to use an enumeration type only in a trivial manner, i.e., just for identification purposes, without exploiting the set of properties of its underlying type.

***Suggestion 2.1***
Java *provides a more advanced way to define an enumeration type as a class type with a predefined set of constant named objects instantiating it. This supports consistent treatment of user-defined types, of which the approach presented in this text would benefit if a similar extension were applied to* C++. *This seems unlikely to happen (soon), though.*

### 2.1.2 Class types

Leaving enumeration types as a speciality to consider only in particular cases, the common way to introduce a new type is to define it as a *class (type)*. Its properties are described by enclosed *member constants*, *types*, *data* and *functions*, whose definitions may depend on each other in this order. The behaviour of the latter may therefore be influenced by the *state* of a particular object of that type, which is given by the state of its *member (data) objects*. In absence of such, the object is *stateless* and its behaviour invariant.

Considering example 2.2, class features which cause the power of object-oriented programming will now be characterized by some of their implications.

⟨**Example 2.2:** `double`-*vector of variable length*⟩≡
```
#include <cstddef>

class VariableDoubleVector
{
public:
    typedef double Element;
    typedef std::size_t Length;
    typedef Element& Reference;
    typedef Element const& ConstReference;
private:
    Element* buffer_;
    Length length_;
public:
    VariableDoubleVector(Length const length=3)
    : buffer_(new Element[length]),
      length_(length)
    {}
    VariableDoubleVector(VariableDoubleVector const&);
    // defined elsewhere
    ~VariableDoubleVector()
    {
        delete[](buffer_);
    }
    VariableDoubleVector& operator=(VariableDoubleVector const&);
    // defined elsewhere
    Length length() const
    {
        return length_;
    }
    ConstReference operator[](Length const i) const
    {
        return buffer_[i];
    }
```

```
        void setElementAt(Length const i, ConstReference element)
        {
            buffer_[i] = element;
        }
    };
```

A probably surprising detail of this example is the provision of merely the `const`-qualified version of `operator[]`, while modifiable random access was replaced by a separate function `setElementAt`. This choice is necessary to avoid the fundamental problem of modifiable access to data structure elements, which is the synchronization of the states of the elements and that of the owning structure, which may depend on each other. The solution requires techniques like smart references that belong to a discussion of data structure design.

### 2.1.3 Class object initialization

A *constructor* and a *destructor* are executed at the start and the end of an object's lifetime to guarantee eventual pre- and post-conditions and to allow to postulate invariants of its state. Because their usage should be familiar to the reader and as this text does not aim at data structure design, just some important properties of constructors shall be resumed.

Constructors are no (member) functions. Unlike these, they may not be referenced by a member function pointer and therefore not be represented by an object. This is on purpose, to guarantee that they are only invoked in a certain order at the beginning of an object's lifetime. To achieve a first-class representation by an object, a separate *factory function* has to be defined instead that delegates its calls to the according constructor.

The call of a constructor may be viewed in a functional sense as the (non-modifying) transformation of its arguments to the constructed object. This is particularly meaningful in the case of a single argument like in example 2.2, where a (non-modifiable) `Length` object may be transformed to a vector: a type `X` is said to be *convertible* to a type `U`

- *implicitly*, if the statement `U u = x;` is legal
- *explicitly*, if the statement `U u(x);` is legal

for an object `x` of type `X`, which introduces an *is-a-relationship* between `X` and `U`. This holds for any of these cases:

- Neither `X` nor `U` has class type and the *standard conversion* rules hold, [ISO98] § 4.
- `U` has a public constructor that accepts an argument of type `X` or of a type to which `X` is implicitly convertible (only explicit conversion is allowed if the constructor is denoted `explicit`), [ISO98] § 12.3.1.
- `X` has a public *conversion operator* to `U`, [ISO98] § 12.3.2.

Thus, whereever possible an object should be initialized explicitly (in *function style*), which on the one hand is guaranteed to work for both kinds of constructors, and which on the other hand uses the same syntax as when calling a constructor that takes several arguments. Finally, this avoids the frequent confusion with an assignment operation. Therefore prefer

```
    VariableDoubleVector v(3);
```

to

```
VariableDoubleVector v = 3;
```

which could and should be enforced by declaring the unary constructor in example 2.2 to be `explicit`. Assuming obedience to this rule, the term *conversion* will be used in the sense of *explicit conversion.* If two types are convertible to each other, both form a common *representation* and are therefore said to *represent* each other.

### 2.1.4 Class interface

Modularity is enforced by *encapsulation* of implementation details of a class by means of *access restriction*. A class type's *interface* accessible from objects of other types basically consists of that set of members that is declared to have `public` access. Exceptions to this rule are the infamous `protected` access mode, which does not truly inhibit public access, and *friend functions / classes*. Both spoil the concept of an interface and should therefore be avoided whereever possible. Frequently, this is achieved by a more precise design of the according data structure, in other cases a clear separation of concerns allows to encapsulate their use within fixed constructs of collaborating classes. While techniques to avoid `protected` access and friend classes belong to a separate discussion of data structure design, reasons for and replacements of using friend functions will be explained in § 7.1.1.

`private` implementation details can be changed without effect on code that uses the class. This benefit is paid by a higher (compared to procedural code) effort to define an interface both general enough to be able to cover many (if not any) implementation alternatives and precise enough to not impose unnecessary overhead which could not be abolished by a well optimizing compiler.

Member objects should always have `private` access, which is indispensable for guaranteeing the validity of the enclosing object's state. Although this seems to be a trivial demand, one finds it frequently violated for the sake of code "simplification". This opens the back door for the disadvantages and dangers of procedural programming. Using access functions instead allows to provide alternative implementations of member access, e.g., to perform access counting or resource locking for thread-safety. On the other hand, their use does not impose any overhead if inlined.

### 2.1.5 Global functions

In contrast to other object-oriented languages, C++ allows the definition of *global functions* which are not members of a class type, like in example 2.3. This allows to provide only such functions as members that actually access member data (e.g., the function `length`), which minimizes the class interface and therefore the scope of code changes.

⟨**Example 2.3:** *Position of the maximal element of a* `VariableDoubleVector`⟩≡

```
VariableDoubleVector::Length maxPos(VariableDoubleVector const& vector)
{
    VariableDoubleVector::Length result(0u);
    for (VariableDoubleVector::Length i=1u ; i!=vector.length() ; ++i)
    {
        if (vector[result] < vector[i])
        {
                result = i;
            }
        }
        return result;
    }
```

***Remark 2.4***
*In example 2.3 and in following examples, it is assumed for simplicity that the vector is not empty, which would have to be guaranteed in a true application.*

Contrary to a class type, the interface of a builtin type does not contain members. Therefore, the (overloaded) global definition of a function allows to call it with either builtin or class type arguments by the same syntax. This holds particularly for operators, which have global syntax even if overloaded by a class member function. In example 2.3, the syntax of operator `<` would not change if the vector elements were of some class type for which this operator is overloaded globally or by a member function.

Note that builtin operators, though, differ from functions overloading operators in not being referenceable code, i.e., they can not be accessed by function pointers.

### 2.1.6 Inheritance

***Problem 2.5***
*As it is impossible to write a single class to describe a particular idea (e.g., that of a vector) that suits all needs, one comes up with several different implementations, each of which resulting suitable just in certain cases. These are semantically related, though, and therefore cause the repetition of code if provided by "monolithic" implementations.*

This effect is reduced by *inheritance*, which is the extension of some *base class* to a *derived class*, possibly adding members. A reduction of the set of accessible members may be simulated by *overwriting* them with equally named members of different kind or with `private` access. This is not a true reduction, though; particularly the set of member objects and therefore the class' size can not be decreased by inheritance. Actually, its application obfuscates the meaning of inheritance, as the derived class is considered to have an is-a-relationship with its base class, which holds syntactically, but not semantically in presence of base member hiding.

***Problem 2.6***
*Therefore, if a function is overwritten by a function of equal name but different signature, it is not accessible via the derived class. If a member function is overloaded to match different tuples of parameter types and overwritten for one of them, each of these versions has to be explicitly overwritten in a derived class to remain part of its interface.*

***Problem 2.7***
*One is forced to overwrite a member function if a reference to the enclosing object is to be returned. In this case, a deriving class has to overwrite that function to return some entity basing on* `this` *to avoid that the object is sliced (see below) to the base class type when returned.*

In example 2.8, a derived class of `VariableDoubleVector` is shown that provides an optimization of the search of the maximal element's position at constant time order, at constant additional costs of the element setting function.

⟨**Example 2.8:** `double`-*vector of variable length with maximal position information*⟩≡

```
class MaxPosVariableDoubleVector
: public VariableDoubleVector
{
    typedef VariableDoubleVector Base_;
private:
    Length maxPos_;
public:
    MaxPosVariableDoubleVector(Length length=3)
    : Base_(length),
      maxPos_(0)
    {}
    // implicit copy constructor
    // implicit assignment operator
    void setElementAt(Length const i, ConstReference element)
    {
        if ((*this)[maxPos_] < element)
        {
            maxPos_ = i;
        }
        Base_::setElementAt(i,element);
    }
    Length maxPos() const
    {
        return maxPos_;
    }
};
```

In example 2.8, a member function `maxPos` is added to return the stored position information, and `setElementAt` is overwritten to actualize this information on modifying element access. If the latter were represented by the non-qualified `operator[]`, it would serve as example of problem 2.6, as the qualified version would have to be overwritten identically just to remain accessible.

Inheritance does not only facilitate the definition of new types, but also induces the implicit convertibility from the derived to the base class type. Therefore, an object of type `MaxPosVariableDoubleVector` may be passed to the function `maxPos` given in example 2.3, as it inherits from (is a) `VariableDoubleVector`. But in that context it would be treated as such, which means that features added in `MaxPosVariableDoubleVector` would not be available. If `setElementAt` were called there, the base class version would be considered, not the overwritten. This effect is called *slicing* and violates the objects integrity.

To obtain the correct behaviour, `maxPos` can be overloaded to take advantage of this optimized vector variant while providing equivalent syntax, as is done in example 2.3.

⟨**Example 2.9:** *Position of the maximal element of a* `MaxPosVariableDoubleVector`⟩≡

```
inline
MaxPosVariableDoubleVector::Length
maxPos(MaxPosVariableDoubleVector const& vector)
{
        return vector.maxPos();
    }
```

In contrast, neither implementation of `maxPos` is applicable to an argument of a type that is not convertible to `VariableDoubleVector`. Consider example 2.10, which is a more storage-efficient alternative to example 2.2 if the number of elements is fixed to 3, e.g., if it stores the coordinates of a point in a 3-dimensional space. This is relevant for performance-critical software, where slight changes in the class code (like storing the object statically on the stack rather than dynamically on the heap) may strongly influence the objects' run-time and/or storage efficiency.

⟨**Example 2.10:** *Fixed length* `double`*-vector (3 elements)*⟩≡

```
    #include <cstddef>

    class Fixed3DoubleVector
    {
    public:
        typedef double Element;
        typedef std::size_t Length;
        typedef Element& Reference;
        typedef Element const& ConstReference;
    private:
        enum
        {
            LENGTH_ = 3
        };
        Element buffer_[LENGTH_];
    public:
        // implicit default/copy constructor
        // implicit destructor
        // implicit assignment operator
        static Length length()
        {
            return LENGTH_;
        }
        ConstReference operator[](Length const i) const
        {
            return buffer_[i];
        }
        void setElementAt(Length const i, ConstReference element)
        {
            buffer_[i] = element;
        }
    };
```

This example demonstrates several inconveniences:

- The class has many literal correspondences with `VariableDoubleVector`, which can not be avoided by plain inheritance, though, as they rely on direct member data access. This is the more unpleasant the less trivial the code is.
- Function `maxPos` has to be provided explicitly for this new type. Besides the type name `VariableDoubleVector` having to be replaced with `Fixed3DoubleVector`, the

remainder of the function code is identical to example 2.3.

- To provide the optimization in example 2.8 for a `Fixed3DoubleVector`, that class has to be replicated as well.

These critiques mainly refer to the question of data structure design, showing that inheritance does not generally solve problem 2.5 yet. But they also illustrate that this is partially due to:

***Problem 2.11***
*An essential bottleneck is the* static binding *of a function with its arguments, which explicitly refers to a particular parameter type to that the according argument type must be (implicitly) convertible.*

### 2.1.7 Performance impact

For an examination of the costs of abstraction, the runtimes of binding the different implementations of function `maxPos` with the according 3-element vector are clocked and displayed in table 2.1. To get a measurable effect, a sufficiently large number of repetitions is performed. The optimizer is hindered from their reduction by the assignment of each call's result to a `volatile` variable, which is required to be performed. The previous examples are compared to a C style implementation that directly operates on a 3-element-array in a length-parameterized function performing a loop like in example 2.3.

As the two latest releases of GCC exhibit certain differences in optimization quality, the results for both are indicated in all experiments to allow to better qualify the results. Advanced optimizing flags are not examined, as the aim of these tests is the comparison under "normal" conditions rather than squeezing the least out of a single solution. Both versions

| **Compiler** | GCC 3.4.4 | | | | GCC 4.0.1 | | | |
|---|---|---|---|---|---|---|---|---|
| **Optimization** | -O0 | -O1 | -O2 | -O3 | -O0 | -O1 | -O2 | -O3 |
| Variable length (2.2 & 2.3) | 17.72 | 4.58 | 4.73 | 5.68 | 18.48 | 4.92 | 5.63 | 5.83 |
| Variable length, adapted (2.8 & 2.9) | 3.42 | 0.30 | 0.30 | 0.30 | 3.66 | 0.30 | 0.22 | 0.22 |
| Fixed length (2.10 & 2.3 analogon) | 16.95 | 3.83 | 5.10 | 5.80 | 17.00 | 3.85 | 4.12 | 6.07 |
| C array & loop | 13.10 | 4.38 | 5.68 | 5.12 | 6.03 | 4.52 | 5.93 | 5.55 |

Table 2.1: Runtimes for explicit binding

behave similarly, actually optimizing worse in various cases if the degree of optimization is increased, being best at `-O1`. This is a frequent effect suggesting not to overestimate the capabilities of the compiler's heuristics.

In general, though, the results indicate that object-oriented programming itself does not impose a performance overhead. The "traditional" implementation by a C array performs even slower than its user-defined fixed-length counterpart, which may be explained by the length of the latter being implicitly provided, which requires only one parameter (a class object reference) to be passed to the function instead of two (a pointer and the length).

The performance of the adapted version is far better—of course, one might say. But later experiments in § 3.2.5 will qualify the inherence of this gain.

## 2.2 Dynamic polymorphism

The term *polymorphism* designates the generalization of type bindings. A typical case is to declare an algorithm to be applicable to objects of different types, instead of just a particular type—while preserving their individual behaviour. This allows to design algorithms just once for the generalization and apply them to objects of any of the types covered by it.

This has the valuable consequence that for the implementation of $D$ different data structures and $A$ different algorithms operating on them, only a constant number of implementations of each algorithm is necessary, which means that code size will be of order $O(D + A)$. In C, one may use a *callback functions* and `void` pointers with many inconvenient/dangerous consequences to obtain this effect. Otherwise, an implementation requires $O(DA)$ code for one algorithm specialization per data structure. This gets even worse for algorithms which operate on several data structures at a time, of which each may be implemented by different versions.

Besides the example of algorithms, polymorphism may be employed in any context where some type is referred, e.g., in the definition of class member objects: a vector could contain objects of any element of a set of types, instead of just a particular type.

### 2.2.1 Virtual functions

object-oriented programming traditionally employs inheritance to realize *dynamic polymorphism*, which is short for: polymorphism by dynamic determination of the object's true type and, accordingly, the functions to be bound. In C++, this is accomplished by declaring as a *virtual member function* any (base) class member function that may have different implementations in derived classes. Its eventual definition serves as default implementation for all derived classes, unless overwritten.

The mere declaration of a virtual function in a base class is legal and called an *abstract function*, which allows to separate the declaration and the definition of a function into different classes. A class containing abstract functions, like in example 2.12, is an *abstract class* of which no object may be created, as well is any derived class which does not overwrite all these declarations by definitions.

⟨**Example 2.12:** *Dynamically polymorphic* `double`*-vector*⟩≡

```
#include <cstddef>

class DoubleVector
{
public:
    typedef double Element;
    typedef std::size_t Length;
    typedef Element& Reference;
    typedef Element const& ConstReference;
public:
    virtual ~DoubleVector()
    {}
```

```
        virtual Length length() const = 0;
        virtual ConstReference operator[](Length const i) const = 0;
        virtual void setElementAt(Length const i, ConstReference element) = 0;
    };
```

The virtue of the declaration of an abstract function is that it provides sufficient information to let the compiler check typing. This way, a function like in example 2.13 can be defined to take arguments of any type derived from the indicated abstract class type, and its implementation therefore can be generalized for arguments of all these class types which only share that formal specification of a subset of their interfaces.

⟨**Example 2.13:** *Algorithm on dynamically polymorphic* `double`*-vector*⟩≡

```
    DoubleVector::Length maxPos(DoubleVector const& vector)
    {
        DoubleVector::Length result(0u);
        for (DoubleVector::Length i=1u ; i!=vector.length() ; ++i)
        {
            if (vector[result] < vector[i])
            {
                result = i;
            }
        }
        return result;
    }
```

No modifications have to be applied to the member function definitions of the previous examples if inheriting from `DoubleVector`, except that function `length` of example 2.10 has to be made non-`static` to overwrite its `virtual` ancestor).

⟨**Example 2.14:** *Dynamically polymorphic variable-length* `double`*-vector*⟩≡

```
    class VariableDoubleVector
    : public DoubleVector
    {
        ⟨⟨like example 2.2, omitting typedefs⟩⟩
    };
```

Similarly, an abstract class `MaxPosDoubleVector` can be derived from `DoubleVector` to declare the virtual member function `maxPos`, by which example 2.9 may be overloaded analogously to hold for any vector implementation by that class.

***Problem 2.15***

*Virtual functions are member functions, which means that their polymorphic effect is restricted to their first argument. In* C++*, there is no analogous construct for global functions to perform a multiple dispatch. While workarounds to this drawback have been proposed, e.g., in [Mey97] § 19, such approaches lack a generalizable principle.*

### 2.2.2 Integration into the type system

Dynamic polymorphism still does not fully solve problem 2.11. The binding of a function with arguments can only be generalized if the types of the latter are references to classes derived from a common base class, in order to apply the virtual function mechanism.

***Problem 2.16***
*This kind of code reuse is inhibited whenever a function should accept arguments of types which are not class references, e.g., builtin types, which have to be bound statically in separate implementations, even if these are literally identical to their polymorphic counterparts.*

But even in pure class context, there is an essential structural drawback of using inheritance from a common base class to formalize the correspondence of type interfaces. One may have noticed that the presence of the (abstract) function `setElementAt` in `DoubleVector` is a too demanding restriction on the parameter type of `maxPos` in example 2.13, as that function is actually not called in this implementation. The latter would perform as well on a non-modifiable vector, which might represent a view onto elements of some other data structure or which might have rvalue elements that are created on demand by some internal factory function. Therefore, a reduced base class of `DoubleVector` should be used to denote `maxPos`' parameter type.

Now, one might as well imagine functions that only require `length` and `setElementAt` to be provided, or only `operator[]` and `setElementAt` (for instance, if the length were implicitly given by some other argument). These would again require the definition of according base classes of `DoubleVector` to appropriately specify their parameter type. Generalizing this situation, one is confronted with:

***Problem 2.17***
*Given some class with $k$ member function signatures, it should be legal to pass an object of this type to any function implementation requiring a subset of them to be provided. On the other hand, such a function's declaration should not be over-restrictive and just require that subset of member functions which is actually called within it. To allow this specification, the class has to inherit a base class that declares exactly those (abstract) functions. Considering all possible function implementations, the class would have to inherit $2^k$ base classes at the point of its definition to thoroughly support this binding method just in case of an according function implementation provided somewhere. As classes typically have at least a dozen or more member functions, this is just illusive.*

*The problem is not proprietary to object-oriented programming and dynamic polymorphism, but occurs in any programming paradigm and method relying on explicit constructs to express* constrained genericity *of polymorphism.*

Usually, the developer reduces the problem domain to obey some hierarchical structure, which may then be reflected by a realistic number of such constraints. While this approach is particularly feasible for large-scale applications, it can not be easily applied to types and functions of low (say, constant) run-time complexity, which on the one hand exist in manifold, hardly hierarchically related variants, and which on the other hand do not allow too costly measures to compensate for differences in their interfaces.

***Remark 2.18***
*There are further binding problems in the presence of* multiple inheritance*, which are not explained here—for being a subject of data structure design—but which are discussed in, e.g., [Str00], [Mey97] § 43 and [Sut02] § 24,26.*

### 2.2.3 Performance impact

The realization of dynamic polymorphism requires some way of access to the true (non-sliced) argument type, at least to the member function definitions provided by it. This can only be obtained if objects of derived class types are treated as base class objects via references or pointers, because slicing would occur otherwise. If a virtual function gets called with a base class reference argument, the actual function to execute (it may have been overwritten for the true type of the referenced object) is looked up and then executed.

This *look-up* is performed at run-time, as references are assigned dynamically, possibly depending on values created during program execution, which coins the term *dynamic binding* for this technique and causes some overhead:

- Each object of a class type having virtual member functions holds a (hidden) pointer to the implementation of these functions appropriate for its true type. As several classes may share the same function implementations, this pointer points to function pointers (stored in the *virtual function table*) rather than directly to the functions' code. This double indirection also copes with multiple inheritance. While the space overhead for the table itself is not relevant, the costs for storing the objects' pointers are prohibitive if (and only if) the objects are small and numerous.

- The run-time overhead per function call is in the range of a few additional instructions, whose effective costs are compiler and hardware dependent. This is too costly for small functions that perform only few instructions themselves and which are frequently called (e.g., in inner loops), but it is as well negligible for nontrivial functions.

- The compiler is able to perform function call optimizations like function call inlining only if the function implementation called is known at compile-time. This refers again particularly to small functions, as inlining larger functions is usually not beneficial.

For a detailed analysis of virtual function call costs see [DH96].

***Problem 2.19***
*While the situation is not critical in many kinds of applications—particularly, if frequent I/O or other expensive operations are involved—it definitely is in most performance-critical software, where huge amounts of small objects are processed by many calls of rather simple functions. In such cases, dynamic polymorphism may cause a multiplication of the costs and is therefore not appropriate.*

Example 2.12 is a worst-case example in which very short functions—that are likely to be executed frequently—are declared to be virtual. This is illustrated by the experiments evaluated in table 2.2, which were performed analogously to their explicitly bound counterparts in § 2.1.7.

While in this case the effect of optimization corresponds mostly with the chosen degree, the absolute times of the most efficient cases have more than doubled against those displayed in table 2.1. In the adapted case they even amount to about six times the previous value, due to the particular simplicity of this implementation compared to the complexity of the virtual function call.

Problem 2.19 was the main obstacle to the consequent adaption of object-oriented programming techniques in performance-critical software where such increases are indisputable,

| **Compiler** | GCC 3.4.4 | | | | GCC 4.0.1 | | | |
|---|---|---|---|---|---|---|---|---|
| **Optimization** | `-O0` | `-O1` | `-O2` | `-O3` | `-O0` | `-O1` | `-O2` | `-O3` |
| Variable length (2.14 & 2.13) | 34.70 | 19.47 | 10.57 | 14.58 | 40.88 | 19.28 | 10.88 | 10.00 |
| Variable length, adapted (2.8/2.14 & 2.9/2.13 analogoi) | 7.70 | 4.15 | 1.65 | 1.73 | 5.25 | 4.08 | 1.65 | 1.48 |
| Fixed length (2.10/2.14 analogon & 2.13) | 33.60 | 19.37 | 10.55 | 9.77 | 21.05 | 19.53 | 10.80 | 9.80 |

Table 2.2: Runtimes for dynamic binding

which lead to a rather conservative programming practice in this field. Many programs are object-oriented just in terms of using objects (for the benefit of data encapsulation), while inheritance is only applied to achieve code reuse, but not to express polymorphism. There is an increasing gap to the actual state-of-art in software engineering, that relies heavily on the power of polymorphism.

***Remark 2.20***
*The virtual function mechanism is still the most efficient solution if the decision on the type to represent some object is not made until run-time or if* storage polymorphism *is required, i.e., if the state of an object of any of a variety of types is to be made permanently available. If one tries to implement some hand-coded polymorphism by* `void`*-pointers, function pointers,* `union`*s and the like, performance losses usually are much worse.*

# Chapter 3

# Generic programming

⟨*implementation*⟩≡
    ⟨*constants*⟩

The term *generic programming* denotes (object-oriented) programming methods that apply *static polymorphism* of data structures and algorithms to gain their maximal combinability and reusability, while avoiding the consequences of problem 2.19. In C++, this approach has been popularized by the introduction of the STL. Its consequent application, though, uncovers that some of the problems mentioned before are of more fundamental nature.

## 3.1 Class templates

A *class template* may be parameterized by types, other class templates, integral (or enumeration) constants, object or function pointers, member pointers or references, [ISO98] § 14.1. Example 3.1 demonstrates that hardly any changes have to be applied to a class to make it *generic*. To explicitly denote a class that is not generic, the term *plain class* will be used.

⟨**Example 3.1:** *Generic variable-length vector*⟩≡
```
#include <cstddef>

template <typename Element_>
class VariableVector
{
public:
    typedef Element_ Element;
    ⟨⟨like example 2.2⟩⟩
};
```

At compile-time, each *instantiation* of a class template by different arguments causes a separate compilation (*implicit specialization*) of the class' source code, with the template parameters being replaced by the arguments. Therefore, example 2.2 corresponds to a specialization of example 3.1 and could be replaced by an alias.

⟨**Example 3.2:** *Replacing a plain variable-length vector by a generic one*⟩≡
```
typedef VariableVector<double> VariableDoubleVector;
```

Each instantiation by a different argument set causes the creation of a new instance, duplicating the entire class template code. There is nothing bad about this—if one would write the code for a `VariableIntVector`, `VariableComplexVector` etc. by hand, one had to implement each of the classes as well, for all being different. Now, it is just the compiler doing the job of code writing.

### 3.1.1 Constant template parameters

Integral types serve to declare not only run-time objects, but as well compile-time constants, particularly formal template parameters. While in the former case a type's value range influences the size of its objects and therefore their run-time memory efficiency, this effect does not play a role in the latter. Thus, for simplicity the type of maximal range can be used for this purpose. To allow to easily change the according choice, `typedef`'ed aliases of arithmetic types are introduced for encapsulation. To underline that these types are guaranteed to be builtin, they are given lowercase identifiers. Though not allowed to serve as template parameters, floating point values may be processed at compile-time as well, see § 4.3.2, and are therefore represented by the type `real`.

⟨*constants*⟩≡
```
typedef bool          boolean;
typedef unsigned long natural;
typedef long          integer;
typedef long double   real;
```
The application of a `natural` template parameter in example 3.3 to define a generic version of example 2.10 is straightforward.

⟨**Example 3.3:** *Generic fixed-length vector*⟩≡
```
#include <cstddef>

template <typename Element_, natural LENGTH_>
class FixedVector
{
public:
    typedef Element_ Element;
    ⟨⟨like example 2.10, omitting the enum definition⟩⟩
};
```
Again, example 3.4 replaces example 2.10 by aliasing the generic implementation.

⟨**Example 3.4:** *Replacing plain fixed-length vector by generic*⟩≡
```
typedef FixedVector<double,3> Fixed3DoubleVector;
```

### 3.1.2 Specialization

After their declaration, templates may be *explicitly specialized* to provide special implementations for particular choices of arguments.

A popular example is that of a `VariableVector<bool>`, as it turns out very inefficient in terms of storage to use the general implementation of `VariableVector` here. Because bits can not be addressed individually, each `bool` value will be stored (at least) as a `char`, usually occupying 8 bits. In a specialization the bits may be packed, which increases access times but decreases storage space drastically, [Str00] § 16.3.11, [Sut02] § 6.

⟨**Example 3.5:** *Explicit complete specialization*⟩≡
```
template <>
class VariableVector<bool>
{
    // specialized definition
};
```

A class template may not only be *specialized completely* for all of its parameters being bound with certain arguments, but also *partially* to just narrow their range. Example 3.6 introduces partial specializations of the class template `FixedVector` that provide an empty vector (the generic implementation would fail, as empty arrays are illegal) and a vector of a single element that avoids the overhead of indirect element access.

⟨**Example 3.6:** *Fixed length vector (0 or 1 elements)*⟩≡

```
#include <cstddef>

template <typename Element_>
class FixedVector<Element_,0>
{
public:
    typedef Element_ Element;
    typedef std::size_t Length;
    typedef Element& Reference;
    typedef Element const& ConstReference;
public:
    static Length length()
    {
        return 0;
    }
};

template <typename Element_>
class FixedVector<Element_,1>
{
public:
    typedef Element_ Element;
    typedef std::size_t Length;
    typedef Element& Reference;
    typedef Element const& ConstReference;
private:
    Element element_;
public:
    static Length length()
    {
        return 1;
    }
    ConstReference operator[](Length const) const
    {
        return element_;
    }
    void setElementAt(Length const, ConstReference element)
    {
        element_ = element;
    }
};
```

Another example of partial specialization of vector class templates is very important for saving compilation time and executable code space: partial specializations for an argument being any pointer type may commonly refer to a single explicit specialization for a `void*` argument, only requiring additional type casts and therefore not imposing run-time overhead, while avoiding the creation of repeated code in instantiations for different pointer types, [Str00] § 13.5, [Mey97] § 42.

## 3.2 Static polymorphism

### 3.2.1 Function templates

Analogously, a *function template* is a parameterized function which is specialized (compiled) separately for each instantiation by a different tuple of argument types. Example 3.7 is a generic alternative to example 2.13 that requires only few changes to the previous version:

- The type of `maxPos`' argument is replaced by a template parameter. Hereby, a new specialization of this code will be created for each instance bound with a different argument type.
- The vector's member types `Element` and `Length` have to be explicitly designated as types by the keyword `typename`. This is necessary because the compiler parses the template code before any of its specializations and expects by default any class member to be a member constant, object or function unless specified to be a type.

⟨**Example 3.7:** *Generic function on vector*⟩≡

```
template <typename Vector_>
typename Vector_::Length maxPos(Vector_ const& vector)
{
    typename Vector_::Length result(0u);
    for (typename Vector_::Length i=1u ; i!=vector.length() ; ++i)
    {
        if (vector[result] < vector[i])
        {
            result = i;
        }
    }
    return result;
}
```

This implementation of `maxPos` could be bound without further adaption with objects of either of the presented vector types, except `FixedVector< ·,0>` of course. In contrast to class templates, the calls instantiating `maxPos` do not have to be explicitly parameterized by the according argument type, if this can be deduced from the argument object's type. It is important to note that it usually does not take much effort to change (well-written) code to rely on static rather than dynamic polymorphism.

### 3.2.2 Function template specialization

Like explicit class template specializations, alternative definitions of function templates may be provided that hold for restricted argument sets or particular choices of them, which permits the implementation of adapted variants and optimizations of algorithms for particular arguments. Those are not considered specializations, though, but overloaded functions, which causes a number of differences to class template specializations with regard to their selection and association. Disregarding them here, an application of function template overloading is demonstrated in example 3.8, whose two latter cases show that a class template instance does not have to be provided by an explicit specialization in order to serve as an accordingly specialized argument type.

⟨**Example 3.8:** *Algorithm specialization*⟩≡
```
template <typename Element_>
inline
typename FixedVector<Element_,0>::Length
maxPos(FixedVector<Element_,0> const&);

template <typename Element_>
inline
typename FixedVector<Element_,1>::Length
maxPos(FixedVector<Element_,1> const&)
{
    return 0;
}

template <typename Element_>
inline
typename FixedVector<Element_,2>::Length
maxPos(FixedVector<Element_,2> const& vector)
{
    return vector[0]>vector[1] ? 0 : 1;
}

template <typename Element_>
inline
typename FixedVector<Element_,3>::Length
maxPos(FixedVector<Element_,3> const& vector)
{
    return vector[0]>vector[1] ? vector[0]>vector[2] ? 0 : 2
                               : vector[1]>vector[2] ? 1 : 2;
}
```
The function template specialization for empty vectors is declared but not defined and will therefore not compile if instantiated. The other cases perform *loop unrolling* and should be much more efficient than the generic version using a loop—unless the compiler applies similar techniques. This demonstrates how hardware-independent optimizations performed by the compiler may be anticipated by standard language code. The extension of these specializations to arbitrary values for `LENGTH_` is possible but more complex to develop, and is recommended to be implemented in a more general manner not discussed here.

### 3.2.3 Concepts

Function `maxPos` as given in example 3.7 is *free*, i.e., it does not formally prescribe a certain parameter type. But the compilation of an instantiation by some argument type will fail unless this type (represents a finite sequence of elements of equal type and) provides:

- a member function `length` which takes no arguments and returns a value (representing the number of elements of the enumeration)
- a member type `Length` that (represents non-negative numbers up to the number of elements in the sequence and) provides:
  - an explicit constructor which takes an `unsigned int` rvalue argument and creates a `Length` object (representing the value of the argument)
  - the infix operator `!=` which takes a `Length` lvalue first argument and the return value of `length` as second argument, and which returns a value that is implicitly convertible to `bool` (and `true` iff the number represented by the left-hand operand is not equal to the number represented by the right-hand operand)
  - the prefix operator `++` which takes a `Length` lvalue argument (and increments by one the number represented by it)
  - the infix operator `=` which takes `Length` lvalue arguments (and sets the state of the left-hand operand to represent the state of the right-hand operand before the assignment)
  - an implicit constructor which takes a `Length` lvalue argument and creates a `Length` object (representing the number represented by the argument before the constructor call)
- the binary operator `[]` , which takes a `Vector_ const&` first argument and a `Length` lvalue second argument and returns a value (representing the element at that position of the sequence which is represented by the argument, count starting from 0) of a type that provides:
  - the operator `<` which takes values of this type as arguments and returns a value that is implicitly convertible to `bool` (and `true` iff the number represented by the left-hand operand is smaller than the number represented by the right-hand operand, requiring the elements to obey a strict ordering)

The conditions set in parentheses are semantical: they do not mean syntactical violations if they are not fulfilled, i.e., they do not cause the compilation to fail, but might lead to wrong results when executed. Such rules are usually expressed in terms of *invariants*, *pre-* and *postconditions*, which may, but do not have to be verified explicitly by evaluating boolean expressions. Additionally, complexity bounds for the evaluation of the denoted expressions may be imposed.

This kind of description of a set of types which are valid to serve as template arguments for a given function implementation, like that of `maxPos`, is called the *concept* to be *modeled* by that argument. In the context of generic programming, the specification of a concept of a template parameter takes the role that a type has in non-generic code, where it specifies the set of legal argument objects for a plain function parameter. The generality of accepting any argument of a certain type is broadened to accepting any argument of any type modeling a certain concept. Hence, a concept may be considered a “type type”. A type specification may be given at different levels of strictness: it may be referred to explicitly, it may have to be convertible to some given type, or it may have to model a given concept.

***Remark 3.9***
*The reader familiar with the* STL *or similar projects might have noticed that the above concept specification is not equivalent to the way it is expressed in these. Besides formal differences, the* STL *style is more explicit on the return types of functions, usually requiring them to model a certain concept or even to be convertible to a certain type. The same holds for the acceptance of argument types. This restriction might seem slight, but will prove significant.*

While the specification of a type is declarative, with an explicitly defined universe of functions applicable to arguments of that type, the specification of a concept is not necessarily so. Up to now, a concept is only given implicitly in C++, just requiring the legality of a function call within a generic context, regardless of any formal specification of the arguments. It is only imposed on demand, i.e., when this function is actually bound with such arguments. This may be expressed formally if types and concepts are understood as sets of those objects which implement or model them:

$$o_1, o_2 \in \mathit{Type} \quad \underset{\not\Leftarrow}{\Rightarrow} \quad o_1 \text{ and } o_2 \text{ have equal interface} \quad \underset{\not\Leftarrow}{\Rightarrow} \quad o_1, o_2 \in \mathit{Concept}$$

where *Type* is some type and *Concept* some concept. Here, the term *interface* is used in a more general meaning than in § 2.1.4, as global functions are also considered part of it. In a generic context, the difference between global and member functions becomes a merely syntactical one.

Concept definition syntax is arbitrary, usually rather informal, which bears the advantage of being basically independent of the programming language used to express the implementation (as long as the language provides all constructs referred to by the concept definition). This does not mean inexactness; concepts are related to types similarly as algorithms are to functions that implement them. To verify them, *concept checking* techniques were proposed and applied in [SL00, WSL01], which consist of an explicit encoding of a concept's properties, allowing to check at compile time whether they are provided by a given type. Rather than producing a compilation fault anywhere within the generic code, checking is performed a-priori at a certain place, resulting in a much more specific indication of the concept not modeled.

***Problem 3.10***
*The concept checking construct is not a predicate, but leads to a compilation error in the case of instantiation by an invalid type. While this is sufficient to control code correctness and to provide clearer error diagnostics, it cannot serve as base for advanced techniques to perform compile-time selection of a valid case out of different, possibly invalid alternatives of functions to evaluate.*

*For instance, concepts do not provide means to specialize the global generic function* `maxPos` *for any vector implementation that provides an adapted* `maxPos` *definition like in example 2.8.*

To cope with this lack, proposals in [SR05, GSW$^+$05] suggest to extend C++ by constructs that allow to specify concepts and to qualify generic functions in order to constrain them to arguments that model given ones of them. As analyzed in [JLSW03, GJK$^+$05], though, these extensions still don't solve a more fundamental problem of explicitly imposing constraints on arguments of a polymorphic algorithm.

The specification of concepts can be simplified by identifying and naming common subconcepts, which are then referred to by their name. Such concept substructuring is used in the definition of the STL, leading to hierarchical structures reminding of class inheritance hierarchies.

Analogously, problem 2.17 comes into play: The domain of a function implementation is defined by indicating one or several independent concepts to be modelled by each of its arguments.Assuming for simplicity that one concept were sufficient to characterize a particular argument, it would be the minimal one (the least derived within the concept hierarchy) which contains all properties required of this parameter within the algorithm definition. Still, the concept will most likely contain more, usually even considerably more properties which the algorithm requires to hold this way, although they are not effectively addressed within its definition.

***Problem 3.11***
*An object of a type which does not model a required concept is not guaranteed to be accepted by an algorithm, even if it provides the full (sub)set of addressed properties. As a consequence, any data structure has to be padded to model some concept in order to serve as argument type for an algorithm which is defined in terms of that concept.*

*In contrast to problem 2.17, in this case a solution is at hand: one might express each property by an individual concept and express a function's prerequisites by a set of such* atomic properties *for each of its arguments. But the above example, though simple, illustrates how properties may rely on each other, which can not be expressed by such an approach if these properties are specified independently.*

Again, this is not just a fundamental critique of the STL approach (which was a main reason not to follow it within this project), but refers commonly to programming paradigms and methods that realize constrained genericity by referring to explicit property specifications.

On the other hand, the definition of STL concepts is strongly dictated by the behaviour of according builtin types, which does not always integrate seamlessly with object-oriented ideas. The random access operator `[]` , for instance, if applied to an object of a type that models the concept of a *random access container*, is expected to return an lvalue, [ISO98] § 23.1.1;12. This surely holds for array and pointer types and may be desired in many cases, but is dangerous or not realizable in others, as it exposes the container's internals and requires the existence of lvalues representing the container's elements.

### 3.2.4 Integration into the type system

#### Builtin types

Global generic functions solve problem 2.16, as they can be bound with arguments of builtin and of user-defined types, because no inheritance from some common base class is required. It is legal to do so as long as the type's interface provides the necessary properties. Here, builtin operators play the role of member functions. Vice-versa, this gives (more) motivation to overload operators for class types, by which these may have interfaces identic to builtin types and be supported by the same algorithms.

***Problem 3.12***
*While classes may inform on related types and constants by containing them as members,*

*there is no such direct association for builtin types, which hinders the common application of (global) generic functions that rely on such information.*

*A solution to problem 3.12 is given in § 6.2.*

### Inheritance

An important peculiarity of template programming in C++—in contrast to Eiffel and Java—is that there is no inheritance mechanism for template parameters. Even if already specialized for some base class (reference) type, a template will be specialized individually when instanciated for arguments of any derived class type. The same happens for types that are convertible to each other (e.g., `int` and `float`). Besides the threat of code bloat, this means that template specializations are not inherited. For instance, if deriving a class from `FixedVector<double,3>`, a call of `maxPos` with an argument of that class type would not be bound to the according explicit specialization given in example 3.8, but to a new, different specialization of example 3.7 for that type.

This may seem a disadvantage. The immense benefit of this fact, though, is that inheritance now solely serves structural purposes, while its semantical meaning of providing an is-a-relationship is not considered anymore. The domains of both techniques are independent and may be combined to powerful collaboration, allowing far stricter decoupling of data structures and algorithms than if just relying on inheritance for both tasks. This proves very influential for software development, as it allows to overcome the hindering restrictions of a fixed hierarchy of class inheritance, and is beneficial to expressing complex type interdependencies.

### Aliasing

Types can be *aliased* by means of the `typedef` keyword, which introduces an alternative name for a type without creating a new type instance.

***Problem 3.13***
*There is no* aliasing *construct for templates in* C++*, which means that each template is individual. Such a construct might appear in a future release of the language standard [SR03], but meanwhile one has to avoid the unnecessary creation of analogous templates in a different way.*

***Solution 3.14 (to problem 3.13)***
*A straightforward* workaround *consists of wrapping a template to be aliased into a class that may be aliased by* `typedef`*.*

## 3.2.5 Performance impact

Each call of the `maxPos` version in example 3.7 with a different argument type forces the compiler to create an according specialization. Within this specialization, function calls that depend on this type may be statically bound, eventually enabling the compiler to perform inlining and any other kind of optimization, thus not implying run-time overhead compared to explicitly bound code.

The effect of static binding is now illustrated similarly to § 2.1.7 and § 2.2.3. Due to problem 3.10, no generic analogy to example 2.8 is provided. Instead, the unrolled `maxPos` version in example 3.8 is compared to an according C style implementation for a 3-element-array argument by an inlined function explicitly unrolling the loop. As a first observation,

| **Compiler** | GCC 3.4.4 | | | | GCC 4.0.1 | | | |
|---|---|---|---|---|---|---|---|---|
| **Optimization** | `-O0` | `-O1` | `-O2` | `-O3` | `-O0` | `-O1` | `-O2` | `-O3` |
| Variable length (3.1 & 3.7) | 21.35 | 5.50 | 5.65 | 5.68 | 18.80 | 6.03 | 6.03 | 5.85 |
| Fixed length (3.3 & 3.7) | 17.82 | 6.37 | 4.67 | 4.58 | 17.47 | 4.42 | 4.58 | 5.92 |
| Fixed length, specialized (3.3 & 3.8) | 12.73 | 0.95 | 0.90 | 0.73 | 10.88 | 0.60 | 0.60 | 0.22 |
| C array & unrolled loop | 4.43 | 1.12 | 0.90 | 0.73 | 4.02 | 0.60 | 0.60 | 0.22 |

Table 3.1: Runtimes for static binding

the expectation is confirmed that there is no significant difference between the performance of the generic and the explicitly bound solution as in table 2.1. As a second, one notices that the unrolled version—regardless whether implemented by a C hack or by a statically bound polymorphic function specialization—performs comparably to example 2.9. Besides the persuasive proof of elimination of the *abstraction penalty*, this case indicates that a data structure optimization like that in example 2.8 does not necessarily bear profit. While performing `maxPos` at equal time, the adapted version comes with additional costs of element setting and member data storage. Therefore, its application is only recommended for considerably longer vectors.

***Remark 3.15***
*One could call the statically polymorphic* `maxPos` *version in example 3.7 with a dynamically polymorphic argument, but the costs would be the same as if using example 2.13. This is because the type of binding (dynamic) is decided in the class, not in the function code.* `maxPos` *respects, but does not decide the way of binding of functions called within. By explicit initialization of the generic* `maxPos` *function with the class* `DoubleVector` *of example 2.12, the according specialization is functionally equivalent to the implementation in example 2.13.*

The advantages of static polymorphism are paid for by more executable code that is generated in comparison to uniquely compiled functions relying on dynamic polymorphism. Therefore, a solution by static polymorphism should be sought if and only if performance or/and type system complexity require it. Otherwise, dynamic polymorphism may and should be employed. There is a vast amount of literature on strategies how to apply the latter. This text, though, focuses right on those cases that call for the former.

# Chapter 4

# Meta programming

⟨*implementation*⟩+≡
  ⟨`If`⟩
  ⟨`Stack`⟩
  ⟨`MakeStack`⟩

A class template can be considered an explicitly defined mapping of the template arguments to classes or to their members. Due to its more general applicability, only the latter alternative will be taken into further consideration. The arguments may be types, class templates, integral or enumeration constants or object or function pointers, member pointers or references, [ISO98] § 14.1. This kind of mapping is evaluated at compile-time, wherefore it does not impose any run-time costs.

Such mappings are said to refer to the *meta stage* of the code, as their outcome forms elements of "conventional" code to be executed at run-time. To specify the latter, the term *soma stage* will be used in the following. The term *stage* refers to the manifold representations of a program, from the time a program is planned until the moment it is executed. This means that there are many more stages and those two are neither the first nor the last of them, but situated at the border between the human-controlled, intuitive part of designing and developing (C++) source code and the machine-controlled, automatic part of creating executable code and executing it. A common goal in software technology is to push this border back to earlier stages of program construction.

Various techniques effectuated at the meta stage have been presented in [Vel95b, CE00], but their integration within conventional run-time code lacks smoothness. Hence, some common approaches are resumed here to be modified in the oncoming to circumvent their inconveniences.

## 4.1 Meta control structures

Rather than to provide run-time code templates, specializations may be used in a more abstract manner, without any run-time semantics. The most popular and important example is the `If` construct, which may be considered an equivalent to the conditional operator `? :` at compile-time.

⟨`If`⟩≡
```
template <boolean IS_VALID_, typename TrueResult_, typename FalseResult_>
struct If;
```

```
template <typename TrueResult_, typename FalseResult_>
struct If<false,TrueResult_,FalseResult_>
{
    typedef FalseResult_ Result;
};
template <typename TrueResult_, typename FalseResult_>
struct If<true,TrueResult_,FalseResult_>
{
    typedef TrueResult_ Result;
};
```

Example 4.1 demonstrates the application of `If` to select a type, based on some constant boolean information. Here it is used to modify example 3.1 to provide a member class template `Reference` instead of `Reference` and `ConstReference`, which provides the according information as a member type `Result`.

⟨**Example 4.1:** *Generic variable-length vector implemented by* `If`⟩≡

```
#include <cstddef>

template <typename Element_>
class VariableVector
{
public:
    typedef Element_ Element;
    typedef std::size_t Length;
    template <boolean IS_CONST_>
    struct Reference
    {
        typedef typename If<IS_CONST_,Element const,Element>::Result& Result;
    };
    ⟨⟨like in example 3.1⟩⟩
};
```

The identifier `If` is confusing, as this name suggests statement behaviour, while the construct is used to form type expressions. It was chosen to be conform with other introductions, though. In fact, differently named alternatives will be introduced in § 9.4.2 and § 10.1.3.

It is important to note the difference between the compile-time and the run-time conditional with respect to their evaluation: While in the latter case either the second or the third operand gets evaluated (possibly causing side effects), depending on whose value being requested, this does not hold in the former case. `TrueResult_` and `FalseResult_` are evaluated before being bound with `If`, therefore both have to be legal type expressions. On the one hand, this restricts the applicability of `If` to valid arguments, as opposed to the run-time variant. On the other hand, this effect might cause the construct to be inappropriate for arguments that are expensive to create, as these (compile-time) costs will always arise for both arguments.

Analogously, further control structures like the switch or the loop (by recursion) were defined in [CE00]. The template specialization mechanism specifies a turing-complete language whose expressions are evaluated at compile-time, which led to the term *compile-time programming* or (*template*) *meta programming* for this technique.

## 4.2 Meta functions

A class template that serves as mapping plays the role of a *meta function*. Depending on the kind of arguments and results (types, constants, functions, or templates), different alternatives of meta function implementations are discussed here.

### 4.2.1 Type arguments

Example 4.2 shows a meta function that maps a type to a type. The `Result` is the type referencing the argument if it is not a reference type, otherwise the argument type itself.

⟨**Example 4.2:** *Meta function returning the argument's reference type*⟩≡

```
template <typename NonReference_>
struct ToReference
{
    typedef NonReference_& Result;
};
template <typename NonReference_>
struct ToReference<NonReference_&>
{
    typedef NonReference_& Result;
};
```

Meta functions like this help to conveniently formulate common type transformations. Its use is demonstrated in example 4.3 that replaces example 4.1, assuming the existence of another meta function `ToConst`. This meta function would have to be specialized for reference types as well, for which the `const` qualifier must be applied to the referenced type.

⟨**Example 4.3:** *Generic variable-length vector implemented by meta functions*⟩≡

```
#include <cstddef>

template <typename Element_>
class VariableVector
{
public:
    typedef Element_ Element;
    typedef std::size_t Length;
    template <boolean IS_CONST_>
    struct Reference
    {
        typedef typename ToReference<typename If
        <   IS_CONST_,
            typename ToConst<Element>::Result,
            Element
        >::Result>::Result Result;
    };
    ⟨⟨like in example 3.1⟩⟩
};
```

Example 4.4 demonstrates the analogous definition of a meta function that maps a type to a constant, indicating whether the argument is a reference.

⟨**Example 4.4:** *Meta function detecting reference types*⟩≡

```
template <typename NonReference_>
```

```
struct IsReference
{
    enum
    {
        RESULT = false
    };
};
template <typename NonReference_>
struct IsReference<NonReference_&>
{
    enum
    {
        RESULT = true
    };
};
```

The disadvantage of this solution is that the type of `RESULT` is proprietary to this template instance rather than one of the builtin constant types, which inhibits a seamless embedding into the general type system, as was announced in § 2.1.1. As long as it is not possible to define typed rvalue identifiers in C++, see § 8.2.3, this has to be accepted and the `enum` value to be presumed convertible to the according type `boolean`, `natural` or `integer`.

In [MC00, Mad, Ale01] many analogous examples of meta functions on type arguments are given.

### 4.2.2 Integral value arguments

Mappings of integral constants can be forced to be evaluated at compile time if the result is requested as a constant expression, e.g., to serve as template argument, constant initializer or array dimension. In combination with recursive class template definition, this permits to define nontrivial meta functions on integral constants which are called at compile-time.

⟨**Example 4.5:** *Meta function returning the factorial of its argument*⟩≡

```
template <natural N_>
struct Factorial
{
    enum
    {
        RESULT = N_ * Factorial<N_-1>::RESULT
    };
};
template <>
struct Factorial<0>
{
    enum
    {
        RESULT = 1
    };
};
```

The meta function given in example 4.5 allows to compute the factorial of a natural value `N` at compile-time as the value of the expression `Factorial<N>::RESULT`. Besides for the applications mentioned above, this value could be used for further compile-time computations or for loop unrolling. When introduced in [Vel95b], the term *meta programming* was coined for such meta functions on integral value arguments.

It is interesting to note that the well-known drawbacks of run-time recursion hold just partially at compile-time. While different meta function calls still increase time and space consumption, this is not true anymore for identical calls. Each template instance, once created, is explicitly kept during the compilation process, wherefore repeated requests for the same instance do not imply its reconstruction, but just a look-up. E.g., if `Factorial<6>` had been computed previously, the computation of `Factorial<7>` would cost only a single additional template instance. The (successful) look-up of `Factorial<6>` is performed anyway, therefore not causing additional costs, as each template instantiation involves a previous look-up to inhibit repeated instantiation of a template by the same arguments.

### 4.2.3 Other arguments

Meta functions on other kinds of arguments, e.g., class templates, may be implemented analogously to those on value arguments. Problems and solutions that will be presented for the latter in § 4.3.1 could be applied to the former two as well, wherefore these are not explicitly introduced here.

### 4.2.4 Meta vs. soma syntax

The syntax of a meta function call is different from its soma analogon, as the alternatives

`Factorial<6>::RESULT` and `factorial(6)`

illustrate. If both the compile-time and the run-time version are implemented correspondingly, algorithm multiplication (by different, but analogous code) is caused. This occurs frequently when complex functions are just composed from simpler functions. For example, the binomial of two integral numbers $n$ and $m$ with $0 \leq n \leq m$ is given by

$$\binom{m}{n} = \frac{m!}{n!(m-n)!}$$

and could therefore be implemented for constant or variable arguments by a compile-time and a run-time version, each calling the according factorial function. (For now, it shall be ignored that this implementation offers pretty bad performance due to the multiple computation of identical products. It is just meant as a simple illustration.)

As well, one might implement versions for $n$ being constant and $m$ variable, or vice versa. Semantically, the 4 versions would be equivalent. They still have to be provided explicitly to satisfy the different syntax requirements, which is undesirable.

***Problem 4.6***
*As a consequence, a solution is sought to make functions "transparent" insofar as they should accept and return both compile-time and run-time objects using the same syntax.*

## 4.3 Meta data structures

The "objects" for which meta functions can be called are all those entities mentioned above that may serve as their arguments or results, i.e., as class template arguments and members. As explained before, this heterogenity as well as the incompatibility with run-time function syntax may be problematic. Hence, an abstraction is introduced by special types without particular run-time semantics (i.e., without member data or run-time functions) that allow the default or copy construction of objects, though. Therefore, they allow to represent meta stage objects by soma stage objects, briefly called *meta objects*. This allows their seamless integration in run-time code, without causing performance overhead. Hence, it is legitimate to call such a type a *compile-time data structure* or *meta data structure*.

Theoretically, it is feasible to provide any kind of (discrete) data structure at compile-time. Here, examples will only be motivated and declared; a modified definition offering an interface compatible to that of run-time data structures will be presented in § 6.1.3.

### 4.3.1 Integral values

Any natural constant (appearing in compile-time computations) can be represented by an unique instance of the class template `Natural`, which is an object type, but which has *meta* rather than soma semantics, i.e., its state is relevant at compile-time, not at run-time. In fact, because the instance is an empty class, the creation of objects of this type may be eliminated by the compiler.

⟨**Example 4.7:** *Meta natural type*⟩≡
```
template <natural VALUE_>
struct Natural
{};
```
Now, meta functions with run-time syntax can be defined by overloading the conventional version for meta natural value arguments.

⟨**Example 4.8:** *Run-time interface for* `Factorial`⟩≡
```
template <natural N_>
inline Natural<Factorial<N_>::RESULT> factorial(Natural<N_>)
{
    return Natural<Factorial<N_>::RESULT>();
}
```
The call `factorial(Natural<6>())` gets evaluated at compile-time and the return type represents the result. This and the argument type being empty classes, the construction of the object returned for formal correctness should not impose any run-time overhead.

To use the result, it is necessary to perform an inverse mapping from the meta value to the according soma (r)value in run-time context.

⟨**Example 4.9:** *Map meta to run-time values*⟩≡
```
template <natural N_>
inline natural metaToSoma(Natural<N_>)
{
    return N_;
}
```
The function given in example 4.9 makes the result of the above meta function call available at run-time as `metaToSoma(factorial(Natural<6>()))`.

***Problem 4.10***
*This is rather uncomfortable, although it should be reminded that this inversion only needs to be performed once for a possibly nested compile-time computation. Still, some means to improve this notation is desired.*

Problem 4.6 remains unsolved: it continues to be necessary to provide separate implementations for run- and compile-time variants of analogous algorithms like that for the binomial, even if they have the same interface. This is because the return type still has to be provided explicitly by a class template.

### 4.3.2 Real values

Floating point constants may not be used as template arguments or member constants. Compile-time computations on them are possible, though, given that operations on floating point constants are evaluated at compile-time. There is no means to force the compiler to do so, as C++ does not have a construct which requires a floating point value to be constant, in contrast to the integral case. But an optimizing compiler should be expected to exhibit the desired behaviour.

Then, a floating point constant may be represented by (a class containing) a function that takes no arguments and returns that value, like in example 4.11.

⟨**Example 4.11:** *Meta representation of $\pi$*⟩≡

```
struct Pi
{
    typedef real Result;
    static Result result()
    {
        return 3.14159265/*...*/;
    }
};
```

Tranforming this to a macro,

```
#define RealDef__(Name__,VALUE__)                                    \
struct Name__                                                        \
{                                                                    \
    typedef real Result;                                             \
    static Result result()                                           \
    {                                                                \
        return VALUE__;                                              \
    }                                                                \
}
```

the definition could be written more conveniently as

```
RealDef__(Pi,3.14159265);
```

Meta functions on such arguments are implemented equivalently, accessing the represented floating point constants by a member function call, as demonstrated in example 4.12.

⟨**Example 4.12:** *Area of a circle with constant radius*⟩≡

```
template <class Radius_>
struct CircleArea
{
    typedef real Result;
    static Result result()
    {
        return Pi::result() * Radius_::result() * Radius_::result();
    }
};
```

To stay within the meta stage, it is necessary to provide the radius as a template argument instead of passing it as a (soma) function argument to `result`.

In [Vel99], the combination of integal and floating point constants in compile-time computations was demonstrated by the definition of meta functions producing the sinus and cosinus of a constant rational subdivision of $2\pi$ as a floating point constant.

Actually, the representation of $\pi$ itself might be deduced from a series expansion of rational constants. This way, explicit definitions of floating point constants could be entirely avoided and instead of being fixed, their value would be determined during compilation up to the representation precision of the machine used.

### 4.3.3 Stacks

As demonstrated in example 4.5, meta programming relies on recursion to perform nontrivial computations. Its natural complex data structure therefore is the recursive accumulation of elements, usually called a *list* (*cons* in LISP). Here, this structure shall be called a *meta stack*, as this describes best its properties.

⟨Stack⟩≡
```
struct Empty
{};
template <typename Front_, class EraseFront_>
struct Stack
{};
```
A stack is formed by nesting `Stack` instances, until an `Empty` argument marks the bottom of the stack. E.g.,

```
Stack<int,Stack<int,Stack<float,Empty> > >
```

is a stack holding the types `int`, `int` and `float`. For the ease of writing, a meta function that comfortably creates stacks up to some maximal number of elements was presented in [Jär99].

⟨MakeStack⟩≡
```
template
<typename Type1_,       typename Type2_=Empty,
 typename Type3_=Empty, typename Type4_=Empty
>
struct MakeStack
{
    typedef Stack
    <   Type1_,
        typename MakeStack<Type2_,Type3_,Type4_,Empty>::Result
    > Result;
};
template <>
struct MakeStack<Empty,Empty,Empty,Empty>
{
    typedef Empty Result;
};
```
Using `MakeStack`,

```
MakeStack<int,int,float>::Result
```

gives an alternative, more handy notation of the previous stack.

The classical stack operations `IsEmpty`, `Front` and `InsertFront` can be provided straightforward on the meta stage like `EraseFront` in example 4.13.

⟨**Example 4.13:** *Meta function that removes a meta stack's top element*⟩≡

```
template <class Stack_>
struct EraseFront;

template <typename Front_, class EraseFront_>
struct EraseFront<Stack<Front_,EraseFront_> >
{
    typedef EraseFront_ Result;
};
```

A meta stack stores types of any kind, including meta objects. This way, multiple meta values can be processed at compile-time by a single algorithm.

### 4.3.4 Other structures

Other data structures that are effective only at compile time can be defined equivalently to the meta stack. For example, in [CE00, Ale01] compile-time lists or vectors were implemented this way, which offer functionalities like random access to their elements. These allow to define linear algebra algorithms that are evaluated completely at compile-time. Quite similar to those is the definition of a meta set. In the literature, though, these container concepts usually are not precisely separated but represented by an universal meta structure that merges properties of those different abstract data types.

Based on these linear structures, non-linear ones can be introduced, e.g., a tree by lists, [CE00], or a matrix by vectors. A very interesting observation is that large parts of their implementation, besides a rather small set of elementary access functions, can be provided analogously to their soma counterparts. Given a solution to problem 4.6, one could use identical components to provide both.

# Chapter 5

# Functional programming

C++ is not a functional programming language, and even though it allows to simulate functional behaviour, its consequent support is inhibited by various language characteristics. For instance, operators are not functions and may not be used as first-class objects.

Anyways, there are some functional constructs and techniques that come into play if generic and meta programming merge with plain soma stage code.

## 5.1 Meta and functional programming

Programming meta functions like `Factorial` in example 4.5 requires different coding strategies than those used for functions in an *imperative programming language* like C++. Meta function implementations obey rules that are familiar from *functional programming languages* like LISP or ML:

- there are no side effects
- objects are provided only as constants or as expressions on objects
- values can not be modified
- loops therefore have to be replaced by recursions

While for many imperative mechanisms it is impossible to provide them at compile time, on the contrary one may use functional notation at run time. Thus, stage-independent implementations have to be expressed by the latter. Specialization allows to provide imperative constructs for the soma case where necessary.

## 5.2 Delayed evaluation

The moment in which a function is bound with (some of) its arguments and the one in which it is evaluated do not need to be identical. Alternatively to the immediate call, a function's evaluation may be *delayed* or *late*, which means that only the source of the arguments is specified, while the determination of their actual values is left to a later evaluation step. For instance, consider the *complete evaluation* of the expression

```
f(x,y,z)
```

where `f` denotes a given function and `x`, `y`, `z` arguments bound with it.

### 5.2.1 Partial binding and partial evaluation

In many applications, at a certain state of evaluation `f` and some of the arguments, say `y`, might be given, while `x` and `z` remain to be bound separately, which might be expressed as

```
f(□,y,□)
```

symbolically. This is an example of *partial binding* of a function, whose result defines a binary function which, if bound with the arguments `x` and `z`, evaluates identically to the complete expression above.

*Partial binding* has the structural benefit, though, that those parts of the implementation of `f` which just depend on `y` might already be *evaluated partially* when partially bound, while only those parts which are influenced by the choice of `x` and `z` remain unbound. If these are bound more than once, e.g., by applying the resulting binary function to each element of a sequence (and some other argument), this *partial evaluation* allows significant optimizations, as invariant computations are not repeated.

### 5.2.2 Formal parameters

The definition of a (partially) unbound function requires some placeholder to refer to a parameter, which was provided intuitively, but not exclusively by $\square$ above. This is usually realized by explicitly assigning an unique identifier, the *formal parameter*, to each of the unbound parameters. These are bound with arguments according to their order in an instanciation of the function. In common procedural and object-oriented languages, one would define a new function

```
name(x,z) := f(x,y,z)
```

of some appropriate ***`name`*** to provide the partially bound function. In a *statically typed language* like C this would be formed like

```
Result name(X x, Z z)
{
    return f(x,y,z);
}
```

to provide the explicit specification of argument and result types. This approach is not really practical to use, though, as even if there is just a single call of a function, its separate definition has to be explicitly provided and a ***`name`*** to be introduced.

### 5.2.3 Lambda expressions

In *lambda calculus*, partial binding is defined by a *lambda expression*

```
λ x z . f x y z
```

indicated by the reserved prefix $\lambda$, followed by the formal parameters, a dot and the definition. This is expressed by some analogous syntax in functional languages and requires the language to be not explicitly typed, like LISP. The resulting function is anonymous, which is fine, as the definition can be placed just where it is used and therefore does not require naming. If the language allows to do so, though, the anonymous function can be used to define a named function.

### 5.2.4 Currying

The term *currying* designates a special variant of partial binding which avoids the declaration of parameter identifiers by assuming a natural order of binding, i.e., by binding the leading parameter(s) and eventually leaving the trailing ones unbound. This simplifies the notation, but restricts the applicability of this method. For instance, the expression

```
f x z
```

binds the first two parameters of `f` with `x` and `z`, rather than the first and the third as intended above. Currying is the consequence of interpreting the expression

```
f x y z
```

as bound from left to right like

```
(((f x) y) z)
```

by which partially bound expressions are defined implicitly within nested parentheses. *FACT!* [Str] and *FC++* [Sma] are C++ libraries which provide mapping types which support currying.

### 5.2.5 Positional parameters

An alternative way of defining a partially bound function can be realized if it is possible to perform index mappings at compile time, which is the case if the language allows meta programming. The formal parameter may then be named implicitly by applying the positional index to a common placeholder like in

```
f(p₁ y p₂)
```

which combines the syntactical simplicity of currying with the flexibility of a lambda expression. This idea was realized in the Boost Lambda Library [JP].

Partial binding is not only caused by leaving a parameter unbound, but also by binding a parameter with a partially bound function. Consider the expression

```
f(p₁ y g(p₂))
```

which binds the third parameter of `f` to an unbound function `g`. This can be interpreted in two ways:

1. The third parameter of `f` is bound with the binary function resulting from delaying `g`, which implies that `f` is a higher-order function and the result of the expression is a unary function.

2. The evaluation of the third argument of `f` is delayed to be bound with the result of `g`, once that this is bound. Therefore, the result of the expression is a binary function, whose second argument is to be bound to the parameter of `g`.

Both versions are feasible, but the second is the natural one, allowing an incremental definition of nested functions. The first interpretation means to terminate that process and to make a delayed function an ordinary first-class argument. Therefore, the property of an expression being delayed shall be considered *transitive*, carrying over to enclosing expressions. A particular operation has to be provided to transform a delayed function argument to a first-class object. Symbolically, this may be expressed by underlining the according delayed expression like

$$\texttt{f(p}_1\texttt{ y }\underline{\texttt{g(p}_2\texttt{)}}\texttt{)}$$

which reflects that this operation is effectuated before the common evaluation process, influencing the interpretation rather than the semantics of an expression.

The existence of such an operation suggests that not all functions are subject to transitive delaying. A function that is automatically delayed will be called a *delayable function*, which is considered to be the default case. A function which does not get delayed and which therefore is immediately evaluated when bound shall be called an *immediate function*.

# Part II

# Techniques

# Chapter 6

# Types

```
⟨implementation⟩+≡
    ⟨internal types⟩
    ⟨meta types⟩
    ⟨convertibility test⟩
    ⟨Traits⟩
```

Thanks to generic programming, the implementation of a function does not have to refer to a particular argument type. Still, an argument's *type category* will play a role. For instance, builtin operators describe mappings on builtin types, and have to be embedded in and reflected by a general system of mappings. To apply them to an user-defined type, though, operators have to be overloaded for such.

Thus, a categorization of types has to be devised to allow the formalization of according decisions. In favor of performance, these should be evaluated at compile-time, hence by template specialization. To allow argument types to be incomplete (if user-defined), the type category has to be detected in a non-intrusive manner, i.e., without making the category information part of the type's semantics. This excludes the use not only of class members for this purpose, but also of techniques like traits, see § 6.2, that require individual specializations for each new type. Instead, the category should be derivable from the mere type declarator.

## 6.1 Categories

### 6.1.1 Builtin types

A *builtin type* is one of the types provided by the C++ language according to [ISO98] § 3.9. The hierarchical structure of type categories defined there is illustrated in figures 6.1-6.3, where a type enclosed within a dashed box is a *qualifiable type* that may be *qualified* by `const` and/or `volatile`, any combination of zero or one of each being called a *cv-qualifier*. A double box indicates a *callable type*, to whose instances the function call operator `()` may be applied. The different categories and variants of builtin types and their characteristics will be discussed in § 6.2 and § 11.1.1.

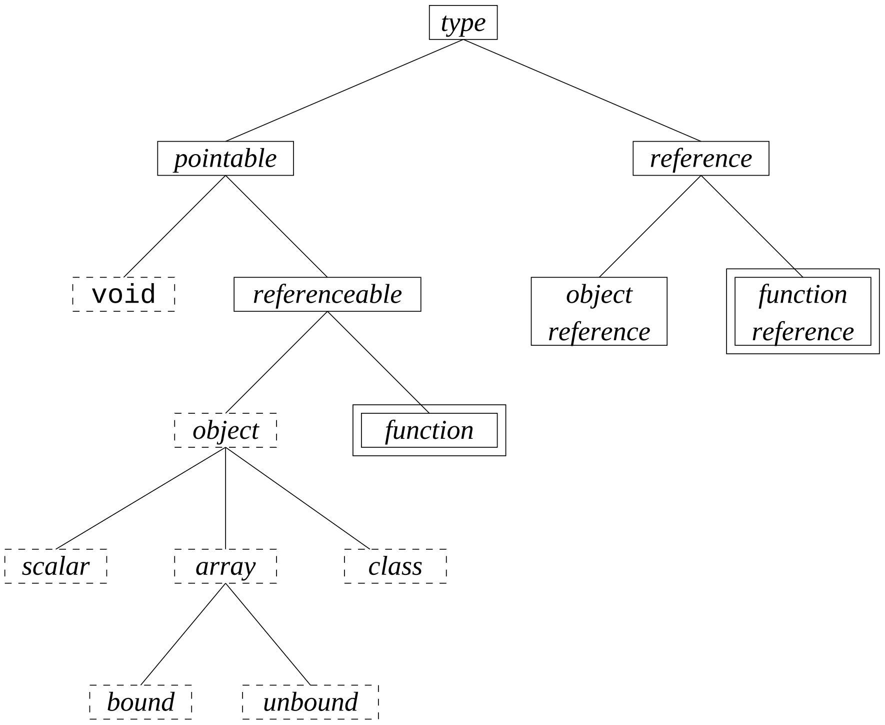

Figure 6.1: Type hierarchy

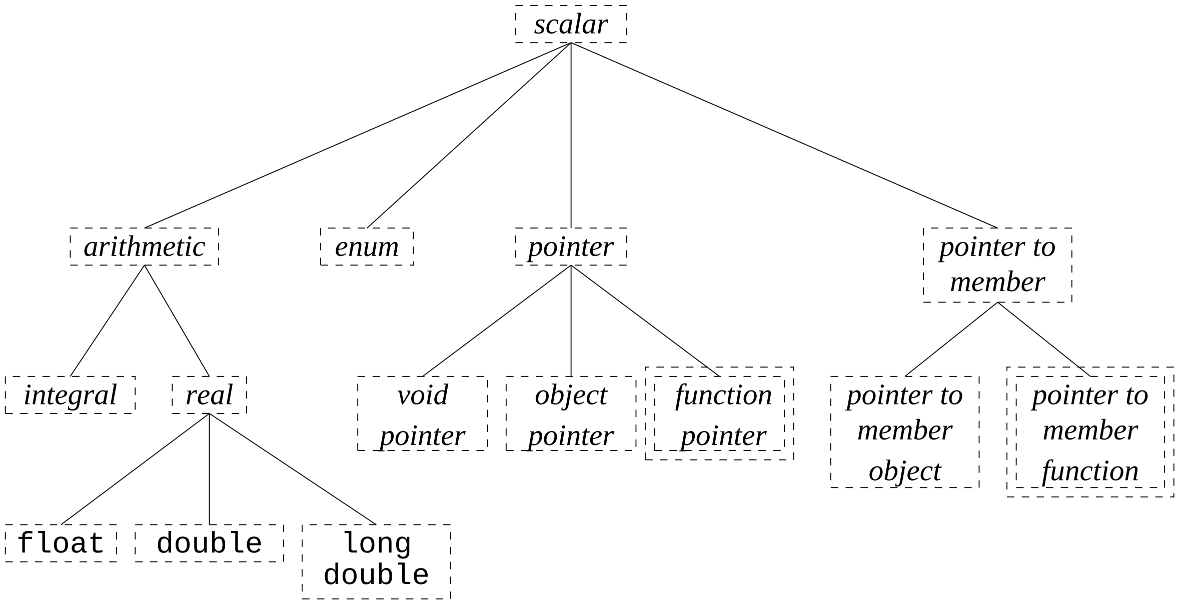

Figure 6.2: Scalar type hierarchy

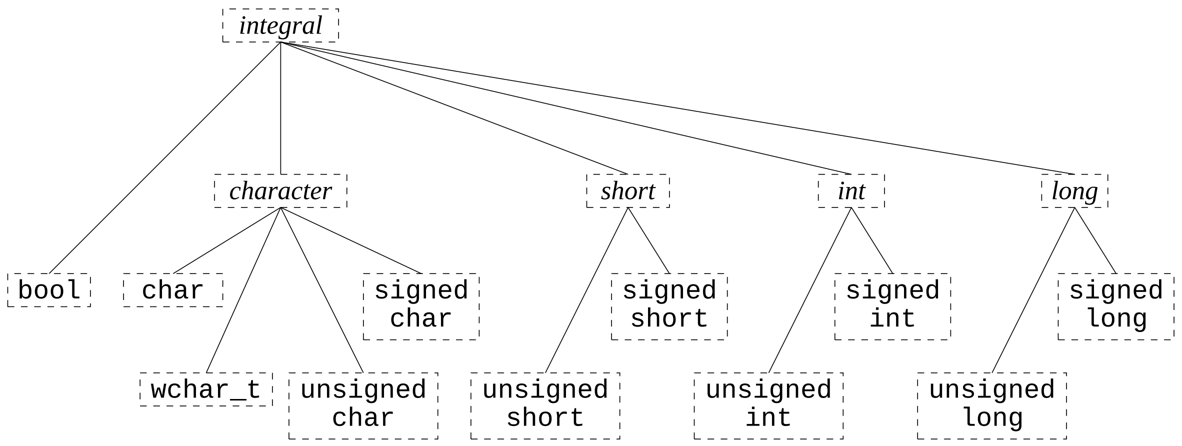

Figure 6.3: Integral type hierarchy

### 6.1.2 User-defined types

Class and enumeration types are left to be defined by the programmer, allowing to specify a virtually unlimited amount of application-specific types. While the question of how to provide an user-defined type shall not be discussed here, some restrictions have to be imposed to allow a categorization. As motivated in § 2.1.1, enumeration types should not be considered user-defined types but just constant sets, as there is no "pure" way of assigning them properties as for class types.

***Remark 6.1***
`union`*s and* bit field*s are not explicitly mentioned in figure 6.1, as they can not be distinguished from plain classes by means of standard* C++ *constructs. Therefore, to be used in a generic context they require hand-tailored adaptation to provide their features by a common class interface.*

### 6.1.3 Internal types

⟨*internal types*⟩≡
  ⟨`Stage`⟩
  ⟨`Binding`⟩
  ⟨`Internal`⟩

Type information that can not be deduced by means of language constructs has to be provided explicitly. As motivated above, to account for possibly incomplete types the type declarators must obey a certain format. Therefore, the applicability of dependent techniques will be restricted to types which are defined based on the knowledge of and in accordance with this approach, called *internal types*.

Any other user-defined type is called an *external type*. There is no formal restriction on this set of types which would allow their explicit detection, wherefore this is the default case if a type can not be identified as element of the previous type categories.

Internal types are identified as such for being an instance of the template `Internal`, which is parameterized by two tags and by a type that provides the effective *specification* of the type's semantics, in a manner depending on its stage.

⟨Internal⟩≡
```
    template <Stage STAGE_, Binding BINDING_, class Specification_>
    struct Internal;
```

The latest stage at which the type's semantics is effective is expressed by the first tag, which takes either of the values `META` and `SOMA` of the type `Stage`. It must be noted that this distinction can not be derived implicitly. It is possible to detect whether a class has data members or whether it is polymorphic, see § 11.1.2, but even in the absence of both a class does not necessarily expose meta semantics.

⟨Stage⟩≡
```
    enum Stage
    {
        META,
        SOMA
    };
```

The second tag is a value of type `Binding`, indicating whether the determination of the type is delayed in the sense of § 5.2 or not. The implications of this differentiation will be discussed in § 9. By default, a type is considered to have `ACTUAL` binding.

⟨Binding⟩≡
```
    enum Binding
    {
        ACTUAL,
        DELAYED
    };
```

### Internal meta types

⟨*meta types*⟩≡
    ⟨Internal *meta type*⟩
    ⟨Integral⟩
    ⟨*meta integral macros*⟩

An *internal meta type* is an instance of `Internal` tagged by `META`. It is just a front-end to its specification which it inherits, thus providing eventual static member information.

⟨Internal *meta type*⟩≡
```
    template <Binding BINDING_, class Specification_>
    struct Internal<META,BINDING_,Specification_>
    : Specification_
    {};
```

The most frequently used case of an internal meta specification is `Integral`, which generalizes the purpose of the meta type `Natural` given in example 4.7 to mapping constant values of builtin integral type to a meta object type.

⟨Integral⟩≡
```
    template <typename type_, type_ VALUE_>
    struct Integral
    {};
```

To simplify the notation of such instances—which would benefit from a template aliasing mechanism—templates and macros are provided that abbreviate the awkward notation of meta integral types and objects.

⟨*meta integral macros*⟩≡
```
template <boolean B_>
struct B
{
    typedef Internal<META,ACTUAL,Integral<boolean,B_> > Result;
};
typedef B<false>::Result False;
typedef B<true>::Result True;
#define b(B_) B<B_>::Result()

template <natural N_>
struct N
{
    typedef Internal<META,ACTUAL,Integral<natural,N_> > Result;
};
#define n(N_) N<N_>::Result()

template <integer I_>
struct I
{
    typedef Internal<META,ACTUAL,Integral<integer,I_> > Result;
};
#define i(I_) I<I_>::Result()
```

***Suggestion 6.2***
*A more consequent approach would be to directly provide constant integral values as meta objects, i.e., to have* `2` *represent the object* `i(2)`*. This would have to be provided by a (non-standard) preprocessor, as there is no means to redefine numerical identifiers in* `C++`*.*

The specification of a *meta stack* has been given by the class template `Stack` and the type `Empty` in § 4.3.3.

## 6.2 Traits

⟨`Traits`⟩≡
```
⟨Undefined⟩
⟨Level⟩
⟨Effect⟩
namespace internal_
{
    ⟨Traits declaration⟩
    ⟨⟨specialization⟩⟩
    ⟨unqualified type dispatch⟩
    ⟨qualified type dispatch⟩
    ⟨reference type dispatch⟩
}
```

Dating from [Mye95], meta functions on type arguments are frequently called *traits*, as their results describe certain characteristics of the argument(s). In a narrow sense, like given in

[Mad, Ale01], they perform a mapping of types to structural properties within the range of the C++ language specification, e.g.:

- the category of the type within the type system, or
- the according qualified, reference or pointer types, if applicable.

In other contexts, though, traits also inform on semantical properties of their argument, like:

- range and precision of arithmetical types (as in [Mye95] or `std::numeric_limits`, [ISO98] § 18.2.1), or
- applicability of functions to objects of the given type and the result of their call (as in `std::iterator_traits`, [ISO98] § 24.3.1).

Traits allow to solve problem 3.12 and to use builtin and user-defined types in generic algorithms equivalently, introducing an additional level of abstraction. As a consequence, semantical traits only need to be defined for those rather elementary properties that apply to builtin types. For example, there is no builtin type to represent an image, wherefore a property describing, e.g., a graphical resolution could surely be provided as a member of the (necessarily) user-defined image class.

Within this section, though, only traits of the first kind are considered, while semantical properties of types will be provided in a different way. In contrast to other approaches, value-based type properties will be represented by the according meta values instead of constants. There are two different approaches to implement traits:

- Stand-alone meta functions like `ToReference` in example 4.2 provide a particular information as a type or value member, usually of some fixed name like `Result` or `RESULT`. This approach—chosen, e.g., in [Mad, Ale01]—has the advantage of being more pure, localizing better the single property mappings. On the other hand, it involves many similar dispatches, as demonstrated in the examples 4.2 and 4.4, where the class template had to be specialized for reference parameters in both cases. Therefore, the implementation is more modular and easier to extend for further properties, but of repeated structure, usually requiring higher compilation efforts.
- A cumulative `Traits` class template provides a set of properties as individually named members. This is the approach of `std::numeric_limits`, `std::iterator_traits` or in [VJ03] and has contrary characteristics to the previous case: It requires all properties to be mapped at the same time, which makes the single traits specializations more longish and complex. Common members of different specializations can be inherited from common base classes. The overall number of templates instantiated will usually be smaller, and one dispatch suffices to cover all property mappings for a particular type category. The introduction of new properties, though, requires to modify all traits template specializations appropriately.

As a consequence, a combination of both strategies shall be applied to express structural traits informing on type properties in the range of C++ features: Independent meta functions defined in § 11.1.1 serve as convenient "front end", but obtain their results from a cumulative `Traits` class template which is not intended for other direct access. Its definition is given by a hierarchical specialization for each type category, resembling the C++ type tree. To select the appropriate specialization from these, a dispatch is used to determine the argument type's category. This process is initiated by a class template `Traits` which will finally inherit the according traits information for its argument.

Before discussing those implementations, a supporting construct is introduced that helps to disambiguate the category of the traits argument type.

### 6.2.1 Convertibility test

⟨*convertibility test*⟩≡
    ⟨`provide`⟩
    ⟨`No` *and* `Yes`⟩
    ⟨`accept`⟩

A technical trick that was suggested in [Ale01] is presented here in a slightly modified form. It allows to use the function overload resolution mechanism at compile-time, which follows different rules than the template specialization selection and therefore allows different specializations and decisions that could not be expressed otherwise. First, a function template `provide` is declared whose return type is the type given as its template argument. Note that the function remains undefined; it just serves to formally provide an expression of this (possibly incomplete or not constructible) type.

⟨`provide`⟩≡
```
    template <typename Type_>
    static Type_ provide();
```

Next element of the technique is the definition of two types that are legal to be used as return types, see [ISO98] § 8.3.5;6, and that are guaranteed to have different sizes. By the following choices, `No` has size 1, while `Yes` has at least size 2.

⟨`No` *and* `Yes`⟩≡
```
    typedef char No;
    struct Yes
    {
        char x[2];
    };
```

The counterpart to `provide` is the declaration of a function template `accept`, an instance of which takes an argument of its template argument type—or of any type implicitly convertible to it. To make tests using `accept` legal for types that don't match, a default alternative is provided that accepts an argument of any type.

⟨`accept`⟩≡
```
    template <typename Type_>
    static No accept(...);

    template <typename Type_>
    static Yes accept(Type_);
```

The alternatives differ in their return type, by whose size the choice can be identified at compile time. As no run-time code is necessary to do so, the functions do not have to be defined, neither will objects of the return type be constructed. Examples illustrating the use of these constructs will be given in § 6.2.4 and § 11.1.2.

### 6.2.2 Properties

Most of the properties that are provided by traits will be introduced in § 11.1.1. Usually, these are types associated with the argument type or boolean constants indicating whether it belongs to a given type category or not. While the latter can always be provided, this does not necessarily hold for the former. For instance, `void` does not have an associated reference type, and function types do not have qualified versions. In order to define a result even for these cases, an incomplete type `Undefined` is declared to indicate invalidity. It serves as default result for all type properties, to be overwritten whenever a meaningful result applies.

⟨`Undefined`⟩≡
```
    struct Undefined;
```

Besides those properties directly emerging from the C++ type system, some are provided that indicate the argument type's role within the extensions given by the approach presented here. This includes the `Binding` property introduced in § 6.1.3, while the `Stage` of an (internal) object is provided by the more general property `Level` expressing an ordering of all principal type categories which reflects the priority by which they will be considered if commonly used within an expression. The meaning of the level `VOID` will be explained in § 7.2.4, while the others are self-explanatory.

⟨`Level`⟩≡
```
    enum Level
    {
        VOID,
        BUILTIN,
        EXTERNAL,
        INTERNAL_META,
        INTERNAL_SOMA,
        UNDEFINED
    };
```

`Effect` describes the way an object of the argument type is evaluated, indicating the extent to which its behaviour depends on the properties of its arguments, if it describes a mapping. A description of the implications will be given in § 9 and § 10; for now it is sufficient to know that the default case is `GENERAL`, particularly if the argument type does not describe a mapping.

⟨`Effect`⟩≡
```
    enum Effect
    {
        IMMEDIATE,
        GENERAL,
        INDIVIDUAL
    };
```

### 6.2.3 Specialization by type category

A previous declaration of the `Traits` class template allows its recursive application within its definition.

⟨`Traits` *declaration*⟩≡
```
    template <typename Type_>
    struct Traits;
```

Now, for each node in the C++ type tree, a traits class template called like that node is introduced which specializes the according information. It inherits the specialization for the according parent node, which allows to locally introduce settings for an entire branch. A non-parameterized root class `TypeTraits` defines default values for all properties.

As a rather interesting case, the traits class template for a reference type argument is outlined. Traits are *reference-transparent*; besides those properties explicitly referring to references, they apply to the referenced type itself, not to referenced types. Therefore, the implementation makes recursive use of the class template `Traits`, where circular dependencies are prevented as it is applied to the non-referenced type which has to be provided by the according argument type dispatch.

⟨`ReferenceTraits`⟩≡
```
template <typename NonReference_>
struct ReferenceTraits
: Traits<NonReference_>
{
    typedef True IsReference;

    typedef NonReference_& ToReference;
    typedef NonReference_ ToNonReference;
};
```

Most of the other `...Traits` classes like `PointableTraits` are implemented in a more straightforward manner, without recursive use of `Traits`. Depending on their specific needs, they may differ in number and kind of template parameters, though.

⟨`PointableTraits`⟩≡
```
template <typename Type_>
struct PointableTraits
: TypeTraits
{
    typedef True IsPointable;

    typedef Type_ ToNonReference;
    typedef Type_* ToPointer;
};
```

Arithmetic type traits provide the type `ToPromoted` to which the type is implicitly promoted, see § 11.1.1. An enumeration type is promoted to the first type of `int`, `unsigned int`, `long`, `unsigned long` into whose range its values fit, [ISO98] § 4.5;2, which is its underlying type. To determine signedness, the explicit cast of a negative value to that type is checked for being interpreted as positive or negative.

⟨`EnumTraits`⟩≡
```
template <typename Type_>
struct EnumTraits
: ScalarTraits<Type_>
{
    typedef True IsEnum;
private:
    typedef typename
            If<(sizeof(Type_)<=sizeof(signed int)),
               typename
```

```
                  If<((Type_)-1<(Type_)0),signed int,unsigned int>::Result,
                  typename
                  If<(sizeof(Type_)<=sizeof(signed long)),
                     typename
                     If<((Type_)-1<(Type_)0),signed long,unsigned long>::Result,
                     Undefined>::Result>::Result
               ToUnderlying_;
    public:
         typedef typename Traits<ToUnderlying_>::ToPromoted ToPromoted;
    };
```

### 6.2.4 Dispatch by type category

Given all those specializations, the appropriate one has to be selected for the given argument type. This is arranged by successive dispatches, which are defined and presented in reverse order of their application.

Assuming that the argument type has been identified as qualifiable by the previous dispatches and given that eventual qualifiers have already been stripped off, the following alternatives remain to be considered:

⟨*unqualified type dispatch*⟩≡
    ⟨`class` *and* `enum` *dispatch*⟩
    ⟨*internal type dispatch*⟩
    ⟨⟨`void` *dispatch*⟩⟩
    ⟨⟨`bool` *dispatch*⟩⟩
    ⟨⟨`char` *dispatch*⟩⟩
    ⟨⟨`wchar_t` *dispatch*⟩⟩
    ⟨⟨`signed char` *dispatch*⟩⟩
    ⟨⟨`unsigned char` *dispatch*⟩⟩
    ⟨⟨`signed short` *dispatch*⟩⟩
    ⟨⟨`unsigned short` *dispatch*⟩⟩
    ⟨⟨`signed int` *dispatch*⟩⟩
    ⟨⟨`unsigned int` *dispatch*⟩⟩
    ⟨⟨`signed long` *dispatch*⟩⟩
    ⟨⟨`unsigned long` *dispatch*⟩⟩
    ⟨⟨`float` *dispatch*⟩⟩
    ⟨⟨`double` *dispatch*⟩⟩
    ⟨⟨`long double` *dispatch*⟩⟩
    ⟨⟨*non-function pointer dispatch*⟩⟩
    ⟨⟨*function pointer dispatch*⟩⟩
    ⟨⟨*member non-function pointer dispatch*⟩⟩
    ⟨*member function pointer dispatch*⟩
    ⟨⟨*bound array dispatch*⟩⟩
    ⟨⟨*unbound array dispatch*⟩⟩
    ⟨*function type dispatch*⟩

Of particular interest is the default case, which covers those types which do not literally or structurally match any of the other cases.

⟨`class` *and* `enum` *dispatch*⟩≡
    ⟨`class` *or* `enum` *dispatch*⟩
    ⟨*default dispatch*⟩

Actually, only `class` or `enum` types are subject to this question. Given a boolean value at compile time indicating whether the argument type is a class, the following dispatch provides the appropriate specialization:

⟨`class` *or* `enum` *dispatch*⟩≡
```
template <boolean IS_CLASS_, typename Type_>
struct ClassOrEnumDispatch_//<false,...>
: EnumTraits<Type_>
{};
template <typename Type_>
struct ClassOrEnumDispatch_<true,Type_>
: ClassTraits<Type_>
{};
```

The default definition of the principal dispatch class template uses the constructs introduced in § 6.2.1 to distinguish between `class` and `enum` by an idea presented in [Ale01] and to provide the first argument to `ClassOrEnumDispatch_`: A null (pointer) is submitted to `accept`, which is explicitly specialized for arguments that form a pointer to member functions of the given type. If the type is a class, it may have member functions, while the null serves as a joker that matches any of them (it does not matter if the class actually has any member function of the specified form). Hence, the specialization is selected. Otherwise the type is an `enum` and may not have member functions. Therefore the specialization can not be applied and the default implementation of `accept` is selected. By the size of the return type the result of the selection is detected.

⟨*default dispatch*⟩≡
```
template <typename Type_, typename Stripped_, typename Pointer_>
struct UnqualifiedDispatch_
: ClassOrEnumDispatch_
<(sizeof(accept<void (Type_::*)(void)>(0)) == sizeof(Yes)),
 Type_
>
{
    enum
    {
        LEVEL = EXTERNAL
    };
};
```

***Remark 6.3***
*According to [ISO98] § 4.10;1, this test can not be implemented by pure meta programming, as the acceptance of converting* `0` *to a pointer to member is based on its (soma) value, not on its type.*

The template `UnqualifiedDispatch_` takes 3 parameters. The first is the (possibly qualified) type itself, the second the unqualified type used to perform the dispatch. The third parameter is a pointer to the second parameter type, allowing another dispatch or a separate analysis to submit additional information to the according traits specialization. In contrast to the dispatch's other cases, those for pointers are kind of interesting, because they base on partial specialization. As a representative, the dispatch for a pointer to binary member function type is outlined here, which translates similarly to the other cases.

⟨*member function pointer dispatch*⟩≡

⟨⟨*0ary member function pointer dispatch*⟩⟩
⟨⟨*1ary member function pointer dispatch*⟩⟩
⟨*2ary member function pointer dispatch*⟩
⟨⟨*3ary member function pointer dispatch*⟩⟩
⟨⟨*4ary member function pointer dispatch*⟩⟩

In this case, the traits specialization expects the result, the enclosing class and a meta stack of the parameter types to be passed as arguments to provide the according traits.

⟨*2ary member function pointer dispatch*⟩≡

```
template
<typename Type_, typename Return_, class Class_,
 typename Param1_, typename Param2_,
 typename Pointer_
>
struct UnqualifiedDispatch_<Type_,Return_ (Class_::*)(Param1_,Param2_),Pointer_>
: MemberFunctionPointerTraits
<Type_,Return_,Class_,MakeStack<Param1_,Param2_>
>
{};
```

The function type dispatch is demonstrated as well by the example of a binary function, the other cases being analogous.

⟨*function type dispatch*⟩≡

⟨⟨*0ary function type dispatch*⟩⟩
⟨⟨*1ary function type dispatch*⟩⟩
⟨*2ary function type dispatch*⟩
⟨⟨*3ary function type dispatch*⟩⟩
⟨⟨*4ary function type dispatch*⟩⟩

Since a template can not be specialized for a function type argument, [ISO98] § 14.8.2.4;9, the additionally passed pointer-to-argument-type is examined.

⟨*2ary function type dispatch*⟩≡

```
template
<typename Type_, typename Stripped_,
 typename Return_, typename Param1_, typename Param2_
>
struct UnqualifiedDispatch_<Type_,Stripped_,Return_ (*)(Param1_,Param2_)>
: FunctionTraits<Type_,Return_,MakeStack<Param1_,Param2_> >
{};
```

Before the presented dispatch can be instanciated, eventual qualifiers have to be stripped off. This is performed by a previous dispatch on the qualification of the argument type, which is almost analogous for all of the 4 possible combinations.

⟨*qualified type dispatch*⟩≡

⟨⟨*non-qualified type dispatch*⟩⟩
⟨`const`-*qualified type dispatch*⟩
⟨⟨`volatile`-*qualified type dispatch*⟩⟩
⟨⟨`const volatile`-*qualified type dispatch*⟩⟩

At this point, the argument is known to be a non-reference type, wherefore this type is pointable and may be passed "pointerized" to the function dispatch, as is demonstrated for the case of `const`-qualification.

⟨`const`-*qualified type dispatch*⟩≡
```
    template <typename Stripped_>
    struct Traits<Stripped_ const>
    : UnqualifiedDispatch_<Stripped_ const,Stripped_,Stripped_*>
    {
        typedef True IsQualified;
        typedef True IsConst;

        typedef Stripped_ ToNonConst;
        typedef Stripped_ const ToNonVolatile;
        typedef Stripped_ ToNonQualified;

        typedef Stripped_ ToStripped;
    };
```
Finally, but first in order of process, reference types need to be covered.

⟨*reference type dispatch*⟩≡
```
    template <typename NonReference_>
    struct Traits<NonReference_&>
    : ReferenceTraits<NonReference_>
    {};
```
This terminates the definition of `Traits`—not completely, though, as properties for certain non-standard type categories introduced within this approach remain to be specified. To cover them, according specializations of `UnqualifiedDispatch_` have to be added. This is done now for those type categories that have already been introduced. While the specialization for `Undefined` does not have to be shown here (the `LEVEL` is set to `UNDEFINED`, the other values do not matter), that for an `Internal` instance is illustrated, although it does not bear surprises either.

⟨*internal type dispatch*⟩+≡
```
    template <typename Type_,
              Stage STAGE_, Binding BINDING_, class Specification_,
              typename Pointer_>
    struct UnqualifiedDispatch_<Type_,
                                Internal<STAGE_,BINDING_,Specification_>,
                                Pointer_>
    : ClassTraits<Type_>
    {
        enum
        {
            BINDING = BINDING_,
            LEVEL = ((int)STAGE_==(int)META ? INTERNAL_META : INTERNAL_SOMA)
        };
    };
```

# Chapter 7

# Mappings

$\langle$*implementation*$\rangle$+≡
    $\langle$*funs*$\rangle$
    $\langle$*binding*$\rangle$

In the previous, rather different approaches were presented which serve to solve some initial problems, but which lack a common structure. To be able to make regular use of them without having to decide again and again on the right technique to use in a given situation, it is essential to find a consistent formulation that embeds the particular methods but unifies them by means of further abstraction. A main interest is the solution of problem 4.6, the embedding of meta into soma code by some uniform syntax.

Ignoring the topic of data structure representation in this place, the more fundamental demand for an alternative way of expressing function definitions will be in focus. This is legitimate because data structure definitions rely on function definitions, while the reverse dependance is possible, but not inherent.

To abstract from the common picture of a function in the sense of C++, subject of discussion will be *mappings*. A mapping

$$\begin{aligned} \mathit{mapping}:\ \mathit{Param}_1 \times \ldots \times \mathit{Param}_n \ &\longrightarrow \mathit{Result} \\ (\,\mathit{param}_1\ ,\ \ldots\ ,\ \mathit{param}_n\,) &\longmapsto \mathit{result} \end{aligned}$$

produces an unique element of the set *Result*, symbolized by *result*, for each given tuple of $n$ *arguments* (or *actual parameters*), which are elements of the sets $\mathit{Param}_i$ and referred to by the (formal) parameters $\mathit{param}_i$. An argument is said to be *bound* with the according *parameter*. For brevity, a mapping as well is said to be *bound* with a tuple of arguments.

One has to distinguish, though, the moment in which a mapping is bound from the (possibly later) one in which it is *evaluated*. This makes an essential difference between the mathematical and the computational concept of a mapping, which is caused by the latter allowing a mapping and its arguments to have a time-dependent state, while the former does not, requiring to translate each state to an individual element of a sequence of states, if there are different ones.

As shown in § 4, *result* and $\mathit{param}_i$ do not necessarily have to be represented by objects nor *Result* and $\mathit{Param}_i$ by types. For instance, the type `int` may serve as argument for a meta stage mapping and as parameter set for a soma stage mapping.

The natural number $n$ is *mapping*'s *arity*. A mapping is called *nullary*/*unary*/*binary*/*ternary*/*N-ary* if $n$ is $0/1/2/3/N$. If the arguments are objects, a *non-modifying mapping* does not change their state. Otherwise, it is called a *modifying mapping*. An unary non-modifying mapping producing a boolean result is called a *predicate*.

A mapping operating on objects is a *first order mapping*. A mapping that operates on other mappings, like `std::sort` does on `cmp`, is called a *higher-order mapping*. A mapping that behaves like an object is a *first class mapping* object.

## 7.1 Functions

The term *function* will be used here in its meaning of the programming language element of a parameterized code block, which holds for both meta and soma functions. The object-oriented programming philosophy of C++ distinguishes a data structure's *interface functions* expressed by member functions from *algorithms* operating on the structure by means of those, expressed by global functions. This separation, though, leads to some common data structure design problems that particularly show up when it comes to complex or special applications.

### 7.1.1 Plain functions

Interfaces of non-trivial data structures are rather rich and involve a considerable amount of member functions. E.g., a vector may provide:

- sequence assignment
- "for each" element algorithm application
- modifying and non-modifying forward and backward iterators
- number of elements
- random read and write access
- element insertion / removal

***Problem 7.1***
*Usually, many member functions are implemented in a generic way in the sense that they are implemented in terms of other interface functions and do not directly rely on any data structure internals.*

*To avoid code replication and to simplify the definition of new data structures, it should be possible to provide interface functions polymorphically, like algorithms.*

For example, in an implementation of `std::vector` or any other data structure offering an STL-style random access iterator, the number of elements may be calculated as the difference of iterators indicating the end and the beginning of the vector.

On the other hand, algorithms are frequently optimized by providing auxiliary data directly within the concerned data structure. An algorithm's result might even be held explicitly as in example 2.8.

***Problem 7.2***
*This contradicts the convention of defining an algorithm by a global function, without allowing it to have access to the class' internal member data. The common workaround is to declare an according specialization of the global function to be a friend function of the class that provides the additional information, which effectively means to make it part of the interface.*

*Hence, any algorithm might have to be included into the data structure's interface to allow eventual optimizations, which breaks with the common idea of interfaces being "minimal".*

According to these considerations, the strict distinction of interface functions and algorithms cannot be held when it comes to realistic applications. Any mapping on a data structure may have to be implemented as either of them. E.g., a binary mapping could be declared as a global function

```
    Result mapping(Param1 param1, Param2 param2);
```

or as a member function

```
    Result Param1::mapping(Param2 param2);
```

As builtin types do not have member functions, the latter syntax applies only if `Param1` is a class type. If `Param1` is not, but `Param2` is a class type (for example, think of the multiplication of a builtin arithmetic type object and a vector), things get trickier. Then, ***mapping*** would most probably be implemented by a friend function of `Param2`.

An alternative approach is to declare it as a member function twin:

```
    Result Param1::mapping1(Param2 param2);
    Result Param2::mapping2(Param1 param1);
```

This seems awkward, but it is a symmetric formulation at least and it is feasible to use it, if a global function ***mapping*** is specialized to call whichever of them is appropriate and hides the inconvenient naming scheme.

***Solution 7.3 (to problems 7.1 and 7.2)***
*Therefore, a mapping should be defined as a global function by default. This allows to forward its calls to an appropriate member function if any argument is of a particular class type or to provide a general definition otherwise.*

***Remark 7.4***
*For the sake of providing the accustomed object-oriented notation, member function calls may be trivially defined to be forwarded to calls of the according global function.*

### 7.1.2 Generic functions

In contrast to a plain function, a generic function is free, which means that its type parameters are not bound with particular types. This *type polymorphism* allows to broaden the scope of a function definition, like to the cases listed below.

On the other hand, using a generic function relaxes the strictness of typing in the function definition syntax: While the compiler surely performs type checking for each instance, it has no means of controlling the correctness of the generic function definition itself. This is analogous to the problem of program verification at the soma stage, for having to consider the entire set of possible inputs to decide on a program's correctness. Consequently, the use of generic functions bears risks which have to be controlled by separate means, see § 7.2.2.

Therefore, generic functions are usually called just in a restricted manner, to model such cases which are rather favourable to abstraction (in the sense of being less likely misused). This again refers mainly to the set of mappings considered algorithms, while in most object-oriented frameworks interface functions are declared to have fixed-type signature, based on the assumption of their low degree of abstraction. For instance, the random access operator `[]` of the STL's `std::vector` takes a parameter of the instance-dependent type `size_type` instead of a generic type parameter.

***Problem 7.5***
*Non-generic implementation, though, prevents interface functions from the common flexibility and performance benefits of generic programming, which would apply to these as well, in spite of or even because of their presumably rather low class of complexity (usually constant). For instance, more efficient code might be generated for random access to a vector given a meta integral parameter rather than a soma one, for its value being available at compile-time.*

***Solution 7.6 (to problem 7.5)***
*All mappings should be defined generically, which goes hand in hand with the decision to abandon the differentiation between interface functions and algorithms as motivated in § 7.1.1.*

Therefore, one would expect any mapping to be provided in a format like:

```
template <typename Param1_, typename Param2_>
Result mapping(Param1_ param1, Param2_ param2);
```

***Remark 7.7***
*The question of passing parameters by reference or by value is not considered here, but delayed to a later discussion in § 8.2.3. Until then, assume the indicated syntax as feasible.*

Like a plain function, a generic function can be overloaded for parameters of particular types or of instances of particular class templates [ISO98] § 14.5.4.2. This somewhat resembles the effect of partial template specialization, as discussed in § 4.2.2, and allows to define specialized versions of the function template besides the general definition for the default case. This approach serves to, e.g.:

- exploit an extended interface offered by an argument to implement the mapping in a more efficient way, see § 7.1.1
- accept both meta and soma arguments, see § 4.3.1
- consider an argument that is given by an indirect representation (*proxy*), like a smart reference, an *expression template* instance [Vel95a] or a delayed mapping (see § 5.2) object
- fully respect the qualification of the arguments, which can only be achieved if each combination of `const` and `volatile` qualifiers is explicitly dispatched, see § 8.2.3

Given a definition that is (generically) overloaded for parameter types which instantiate a certain class template, the C++ overload resolution mechanism, [ISO98] § 14.5.5.2, 14.8.2-3, performs syntactical matching of the argument type against that template class to decide whether the definition is a candidate. The usual consequence is to have the parameter type's properties in question to be reflected within its name, very much like a type's category was decided to be indicated in § 6.1.3.

***Problem 7.8***
*A consequent application of this approach leads to a complicated type declarator syntax which expresses all of the type's properties by according template classes and template parameters. While this is not a substantial problem, in contrast to the category properties the set of semantical properties is not limited, wherefore the notation would have to extended for each new application domain to be covered.*

In the case of multiple matches, the template specialization selection mechanism refers to a partial ordering of the matches, [ISO98] § 14.5.5.2, in terms of the single template argument types being more or less specialized, which again depends on their syntactical properties [ISO98] § 14.8.2.

***Problem 7.9***
*On the one hand, this ordering does not necessarily coincide with the intended priorities depending on the parameters' semantics. For instance, to specialize a unary function template to consider different properties of its argument at the same time—like its stage on the one hand and its representation on the other—the implementor has to provide one specialization for each possible combination of properties to guarantee an unique best match, which results in an exponential number of specializations.*

***Problem 7.10***
*On the other hand, similar difficulties arise if just a single property is considered, but if the function takes multiple parameters, each of which exposing that property. This problem is well-known, e.g., from the implementation of generic functions that are capable of taking expression template parameters [Vel01, RHC⁺96].*

The implicit selection mechanism provided by C++ is insufficient to model behaviour based on semantical type properties. Therefore, other parameter properties than those fundamental ones provided by the type category information should be subject to explicit evaluation that is capable of expressing more complex dependencies, which in turn should lead to the call of a separate function providing the appropriate definition. The evaluation can be expressed by calls of according meta functions, given some kind of explicitly available encoding of the type properties, e.g., by type members if the type is a class. This corresponds with the principle of object-oriented programming and allows to inherit such property information.

In contrast to properties depending on a type's semantics, category properties exist only in a fixed number and possess a certain hierarchy, wherefore the number of combinations of them is manageable and many of them may be excluded rightaway from consideration.

***Solution 7.11 (to problems 7.9 and 7.10)***
*Therefore, any mapping is provided as an* unconstrained generic function *which formally accepts any argument type. It is overloaded to implicitly detect any feasible combination of category properties of its argument types, which follows a static pattern, though, and can therefore be implemented in either general or automatically specialized manner. The support of semantical properties is left to particular specifications of mappings and types.*

## 7.2 Funs

```
⟨funs⟩≡
    ⟨Void⟩
    ⟨Exp⟩
    ⟨UndefinedFun⟩
    ⟨MapSpecExp⟩
```

Motivated by the previous considerations, the suggested approaches to generalize mapping representations will now be formalized, while providing solutions to further technical or design problems.

### 7.2.1 Result type information

In contrast to the assumptions made within the STL, the return type of a generic mapping is not necessarily fixed, but may depend on the respective argument types. Its explicit availability is necessary, though, to define a variable to store the returned result

```
template <typename Param1_, typename Param2_>
Result$_{mapping,Param1\_,Param2\_}$ mapping(Param1_ param1, Param2_ param2);

Result$_{mapping,double,char\ const*}$ x = mapping(3.14159265,"pi");
```

or to use it to define another mapping:

```
template <typename Param_>
Result$_{mapping,int,Param\_}$ mappong(Param_ param)
{
    return mapping(0,param);
}
```

***Problem 7.12***
C++ *does not provide explicit information on the return type of a function call, although there is an proposal to add to the language an operator* typeof *[JS04] which provides the type of any expression and which is already supported by language extensions of* GCC.

***Solution 7.13 (to problem 7.12)***
*A common approach is to provide the result type by a meta function that is named analogously to the concerned function.*

```
template <typename Param1_, typename Param2_>
struct Mapping
{
    typedef ... Result;
};
```

Now the function's prototype looks like:

```
template <typename Param1_, typename Param2_>
typename Mapping<Param1_,Param2_>::Result
mapping(Param1_ param1, Param2_ param2);
```

and the return type is at hand for explicit usage as in the above examples. This construct provides *static reflection* on generic function calls, i.e., a means to obtain and process information about properties of the code itself, which in contrast to common approaches (like Class objects in Java) is given at the meta stage.

***Problem 7.14***
*Unfortunately, such usage requires the duplication of the function call notation: one to obtain the result type, one to perform the call itself. This gets even worse if the result type of a generic function is to be retrieved itself as the* typeof *an expression based on the arguments—then the entire function definition may have to be reproduced in the return type declaration.*

The use of meta object return types allows to avoid the use of member constants, though. The common reason to provide these is to be able to use them as compile-time constant expressions, e.g., to specify the length of an array. This can not be achieved by a soma type function return value, which always has run-time semantics even if it is known at compile time. Using a meta integral value as return type, the information is preserved at the meta stage.

### 7.2.2 Constrained genericity

A first step to overcome problem 3.11 is to break up concepts into *atomic subconcepts* which can not be further subdivided. This complicates an algorithm's specification, as a set of them has to be listed to describe each parameter type's prerequisites, but gets rid of the abundance of type specification.

What does an atomic concept contain? To start with the final answer: a single mapping. This is motivated by the following examination of the kinds of an `STL` concept's syntactical contents (see § 3.2.3):

- *Mappings* may be distributed to separate concepts, as there is no interdependence between mappings within a concept. Each mapping is defined generically, as postulated in § 7.1.2. Its return type is given in some way like the one suggested in § 7.2.1 for any type modelling this concept. Instead of providing explicit definitions of its parameter types, these are generic and required to model according concept sets.

- *Constructors* are not mappings, but they may be encapsulated within (static) *factory mappings*.

- *Types* serve to specify parameters or results of mappings. As a consequence of the previous, there is no further need to provide them separately, as they are part of the mapping specifications.

- *Constants* are replaced by mappings returning them as meta objects.

- *Space complexity bounds* may be provided by mappings which encode complexity classes as meta objects.

- *Invariants* may as well be expressed by meta predicates.

Therefore, each atomic concept can be identified with the specified mapping, which provides its result type, its implementation and its semantical description. This inverts the association of binding validity: Rather than having a concept specify the set of all mappings which may be bound with a data structure modelling the concept, a mapping itself (implicitly) specifies all parameter type tuples which it binds.

Given solution 7.13, an alternative way of concept implementation that solves problem 3.10 is at hand:

***Solution 7.15 (to problem 3.10)***
*For invalid bindings of a mapping, its result type is defined to be the special type* `Undefined` *that is reserved to indicate this case. Meta functions can retrieve the result type for a given parameter type tuple and check whether it is equal to this type to detect illegal bindings.*

***Remark 7.16***
*This approach allows to provide alternative definitions of a mapping and to check their validity for a given parameter type tuple by "trying them out" at compile-time. If there is an unique valid match, it is chosen as the appropriate implementation, otherwise this binding of the mapping is un- or overdefined and therefore invalid, without necessarily causing a compilation failure, though.*

***Remark 7.17***
*The validity of a mapping's binding depends of the validity of the bindings of all mappings that constitute its implementation and can therefore be derived straightforwardly from those by an appropriate meta function. This is not lined out in detail at this point, as it will be subject to further discussion.*

### 7.2.3 Multistage functions

Remark 7.17 and similar previous observations illustrate that each overloading definition of a function template *mapping* to implement a mapping will have to be accompanied by an analogous specialization of the class template *Mapping* to provide the according return type. This is particularly inconvenient for the rules of overload resolution and specialization matching being similar, but not equivalent. To avoid this duplication, both the function and its `Result` type are defined in the same class template which may be specialized.

```
template <typename Param1_, typename Param2_>
struct Mapping
{
    typedef ... Result;
    static Result result(Param1_ param1, Param2_ param2);
};
```

The implementation of *mapping* is now reduced to a mere forwarded function call, which can be inlined.

```
template <typename Param1_, typename Param2_>
inline
typename Mapping<Param1_,Param2_>::Result
mapping(Param1_ param1, Param2_ param2)
{
    return Mapping<Param1_,Param2_>::result(param1,param2);
}
```

***Remark 7.18***
*A member function of a class template is considered a template itself and is therefore evaluated lazily.* `Mapping` *may be instantiated and its* `Result` *be obtained even if the definition of* `result` *is not legally formed. This is particularly important in the case of the* `Result` *being* `Undefined`*.*

This way, the meta function *Mapping* and the generic soma function *mapping* are merged to form a *multistage function*, which in contrast to a meta function does not only produce a `Result` type, but also a function `result` that creates an object of this type, given arguments of the specified parameter types.

The instantiation of the *Mapping* class template may be viewed as the binding of that mapping at the meta stage. The soma function call of `result` remains unbound, until appropriate argument objects are provided and bound at the soma stage.

### 7.2.4 Void arguments

***Problem 7.19***
*In C++, the type* `void` *is used to specify "no type" in function parameter type or return type specifications. Given a* `void` *function* `f` *and a function* `g` *returning* `void`*, the composition* `f(g())` *is illegal, though, albeit it may be interpreted as the sequential but independent call of both functions. This case occurs in (generic) higher-order mappings, e.g., function composition, and has to be covered by a specialization.*

***Solution 7.20 (to problem 7.19)***
*An object type* `Void` *is defined and used as return type instead of* `void`*. It has meta semantics and should therefore not cause any run-time costs. In the mathematical sense, this type represents the empty set at the meta stage and its objects at the soma stage.*

⟨Void⟩≡
```
    namespace internal_
    {
        struct Void_
        {};
    }
    typedef Internal<META,ACTUAL,internal_::Void_> Void;

    #define _ Void()

    namespace internal_
    {
        ⟨traits for Void⟩
    }
```
The underscore identifier `_` abbreviates the notation of expressions involving `Void` objects, which have to be respected within the type categorization traits.

⟨*traits for* Void⟩≡
```
    template <typename Type_, typename Pointer_>
    struct UnqualifiedDispatch_<Type_,Void,Pointer_>
    : ClassTraits<Type_>
    {
        enum
        {
            LEVEL = VOID
        };
    };
```

***Problem 7.21***
*The implementation of a function that takes a variable number of parameters usually has to be solved either by means of default parameters—requiring their type to be default constructible and causing according costs on each function call—or by providing overloaded definitions for*

*the function taking 1,2,3,... parameters, which again requires to provide according specializations within a generic context.*

***Solution 7.22 (to problem 7.21)***
*Things get easier for mappings, as they allow to influence the parameter types of the contained function. Parameter type lists of arbitrary arity are padded to their maximal length at the meta stage by the default parameter type* `Void`*. This does not introduce unnecessary costs for calls of the contained function at the soma stage, which is defined for the maximal number of parameters.*

Given this, it is recommended to treat mappings of a fixed parameter number the same way, to allow to treat *all* mappings uniformly within mappings operating on them. The maximal arity is assumed to be 4, which could be augmented in a straightforward manner in all the constructs that are to be presented.

```
template <typename Param1_, typename Param2_,
          typename Param3_=Void, typename Param4_=Void>
struct Mapping
{
    typedef ... Result;
    static Result result(Param1_ param1, Param2_ param2, Param3_, Param4_);
};
```

The soma function `result` has to be bound with the according number of arguments, which is no problem, though, as these are padded by the forwarding function call.

```
template <typename Param1_, typename Param2_>
inline
typename Mapping<Param1_,Param2_>::Result
mapping(Param1_ param1, Param2_ param2)
{
    return Mapping<Param1_,Param2_>::result(param1,param2,_,_);
}
```

***Remark 7.23***
*An alternative approach is to pass the arguments within a tuple [Jär99]. This is not followed here with intention, as a tuple is an user-defined soma type, whose definition will—according to this approach—have to rely on mappings. This circular dependency must be avoided, which does not prevent to introcuce the tuple style, though, once that the underlying constructs (including mappings) are defined.*

### 7.2.5 Meta expressions

In analogy to calling a bound soma function a *(soma) expression*, a bound meta mapping may be considered a *meta expression*, which is a class that provides a member type `Result`.

The simplest example is `Exp`, which just embeds a given type within a meta expression, providing the type as its `Result`. It serves as an *atomic expression* to form the leaves of a meta expression tree.

⟨Exp⟩≡
```
    template <typename Result_>
```

```
struct Exp
{
    typedef Result_ Result;
};
```

***Remark 7.24***
*Although a meta expression is a complete type, it is only intended to serve as a meta object, not as a soma object type, i.e., to construct soma objects. This could be expressed by declaring its constructors to be private, which is omitted here, though, for the sake of simplicity.*

If a meta expression also provides a member function `result` with soma stage semantics, it is called a *soma mapping meta expression*.

### 7.2.6 Specification

According to solution 3.14, the class template representing a mapping should be wrapped into a plain class to allow to alias the mapping. This also facilitates to provide the mapping as a type template parameter, as dealing with template template parameters is rather inconvenient.

```
struct FunSpecification
{
    template <typename Param1_, typename Param2_,
              typename Param3_=Void, typename Param4_=Void>
    struct Call
    {
        typedef ... Result;
        static Result result(Param1_ param1, Param2_ param2, Param3_, Param4_);
    };
};
```

A class type of the suggested layout is a *specification*! specification of a mapping that is stateless at the soma stage, i.e., its behaviour is invariant at run-time. This specification may serve to declare the according mapping as an internal type

```
Internal<META,BINDING,FunSpecification>
```

Such a type shall be called a *fun type* and its objects *fun objects* or just *funs*. In fact, not every fun type is given in such explicit format, but many are defined in an indirect manner. Therefore, any invariant multistage mapping type is called a fun type.

The first example is `UndefinedFun`, which is what its name says. It is not intended for external use, but will serve the definition of other mappings.

⟨`UndefinedFun`⟩≡
```
namespace internal_
{
    struct UndefinedFunSpec
    {
        template <typename Param1_, typename Param2_,
                  typename Param3_, typename Param4_>
        struct Call
        {
            typedef Undefined Result;
            static Result result(Param1_, Param2_, Param3_, Param4_);
        };
    };
}
typedef Internal<META,ACTUAL,internal_::UndefinedFunSpec> UndefinedFun;
```

## 7.3 Maps

### 7.3.1 Specification

A *map* is a mapping with soma semantics, which means that it may contain and process member data and therefore have a state at run-time. For instance, a map representing an algorithm that uses nontrivial temporary data could provide this data permanently, so that it would not have to be constructed and destructed for each application of the algorithm.

This has the consequence that a map object has to be available within the definition of its evaluation function and therefore to be provided as its 0th parameter. To avoid confusion with funs, this template is called `Apply` rather than `Call`.

While the layout of a map specification—which has to appropriately support member data—depends on the style of data structure implementation and shall not be discussed here, the part essential for providing mapping behaviour looks as follows:

```
struct MapSpecification
{
    ...
    template <typename Param0_,
              typename Param1_, typename Param2_,
              typename Param3_=Void, typename Param4_=Void>
    struct Apply
    {
        typedef ... Result;
        static Result result(Param0_ param0,
                             Param1_ param1, Param2_ param2,
                             Param3_, Param4_)
        {
            ...
        }
    };
    ...
};
```

The presence of the function call operator `()` , though, is not a characteristic of a map, but just a technical addition which can be implemented generically.

### 7.3.2 Fun embedding

***Problem 7.25***
*It should be possible to bind the same higher-order mappings with funs and maps, in spite of the layout differences of their specifications.*

***Solution 7.26 (to problem 7.25)***
*One might just decide to upgrade any fun specification to a map specification and always provide a* 0*th parameter. This is not necessary, though, as problem 7.25 only occurs at some determined cases, which can be explicitly covered by locally wrapping the fun specification into a map specification which takes a dummy* 0*th parameter.*

To better match future needs, not a type is wrapped by another, but a fun specification meta expression is wrapped by a map specification meta expression of the name `MapSpecExp`. The construct is not intended to be used externally, though, but just to ease some elementary definitions.

```
⟨MapSpecExp⟩≡
    namespace internal_
    {
        template <class FunSpecExp_>
        struct MapSpecExp
        {
            struct Result
            {
                template
                <   typename Param0_,
                    typename Param1_, typename Param2_,
                    typename Param3_, typename Param4_
                >
                struct Apply
                {
                private:
                    typedef typename FunSpecExp_::Result
                            ::template Call<Param1_,Param2_,Param3_,Param4_>
                            Call_;
                public:
                    typedef typename Call_::Result Result;
                    static Result result(Param0_,
                                         Param1_ param1, Param2_ param2,
                                         Param3_ param3, Param4_ param4)
                    {
                        return Call_::result(param1,param2,param3,param4);
                    };
                };
            };
        };
    }
```

## 7.4 Binding

⟨*binding*⟩≡
    ⟨`Bind` *declaration*⟩
    ⟨`Dispatch`⟩
    ⟨`Bind`⟩
    ⟨`Fill`*N*`__`⟩

Depending on its arguments' type categories, a mapping's implementation will have to be chosen out of several alternatives. This selection process is analogous for all mappings and takes place before evaluating the individual mapping definition. Therefore an uniform implementation performs these decisions for any kind of mapping.

### 7.4.1 Generalization

In § 7.2, it was not discussed how to call a fun, and for a map this is not clear either. Mappings may be used multiply, but are defined just once, wherefore it makes sense to design them in a way that eases the notation of their use rather than that of their definition. Given funs `F`, `G` and `H`, it would be nice to be able to write meta stage bindings like:

```
F< G<a,b>, H<c> >
```

This requires that funs are not bound with type parameters, but with bound funs and the like, i.e., with meta expressions, which provides a much more convenient notation in comparison to

```
F<G<a::Result,b::Result>::Result,H<c::Result>::Result>
```

To allow conform treatment of all mappings, this way of binding is generally adopted.

### 7.4.2 Anonymous binding

To deal with mappings uniformly, their binding has to be implemented in a way that accepts the concerned mapping—which might be a map—as a generic parameter. This leads to a generalized formulation of the binding of a mapping:

$$\begin{aligned} \mathit{bind}: \ \mathit{Mapping} \times \mathit{Param}_1 \times \ldots \times \mathit{Param}_n &\longrightarrow \mathit{Result} \\ (\,\mathit{mapping}\ ,\ \mathit{param}_1\ ,\ \ldots\ ,\ \mathit{param}_n\,) &\longmapsto \mathit{result} \end{aligned}$$

It is implemented by a class template `Bind` taking (up to) 5 arguments, of which the first (indexed by `0`) is the type of the mapping to be evaluated and the further ones are the argument types passed to it, all given as meta expressions.

⟨`Bind` *declaration*⟩≡

```
    template <class Exp0_,
              class Exp1_=Exp<Void>, class Exp2_=Exp<Void>,
              class Exp3_=Exp<Void>, class Exp4_=Exp<Void> >
    struct Bind;
```

Instances of `Bind` will have the well-known members `Result` and, if the mapping is a soma mapping, `result`, which is 5ary according to § 7.3.2. Its notation as

```
Bind<Exp0,Exp1,Exp2,Exp3,Exp4>
```

at the meta stage and

```
Bind<Exp0,Exp1,Exp2,Exp3,Exp4>
::result(arg0,arg1,arg2,arg3,arg4)
```

at the soma stage, where `arg`*N* is an object of the type `Exp`*N*`::Result`, resembles that of expressions in functional languages. For example, in LISP—which is untyped and therefore does not need nor allow a meta stage—the according expression would read analogously

```
( arg0 arg1 arg2 arg3 arg4 )
```

### 7.4.3 Analyzation

When `Bind` is instantiated, its arguments have to be analyzed in order to decide in which manner the specified mapping is to be applied, as there will be type category dependent ways to do so. For this purpose, `Bind` instanciates a separate dispatching class template that receives the arguments in an appropriately resolved form.

***Remark 7.27***
*Instead of a single dispatch, which will prove rather complicated, a layered approach might be used, which analyzes the properties one by one in hierarchical order by separate dispatches. This would introduce quite a number of additional templates, though, and therefore increase compilation efforts. On the other hand, the design of the dispatch is an internal matter that is invisible to the user, so that its rather complicated structure and use only affects those who should be affected.*

So far, `Bind` has been nominated to be the entrance point for the evaluation of a mapping and traits as the argument type analyzation tool. Now, the central part of the evaluation procedure can be defined, which is the class template `Dispatch` that takes the arguments in analyzed form and selects the appropriate implementation of the mapping to be evaluated.

```
⟨Dispatch⟩≡
    template <Level LEVEL_, boolean IS_DEFINED_, boolean DELAY_,
              typename Stripped0_,
              typename Stripped1_, typename Stripped2_,
              typename Stripped3_, typename Stripped4_>
    struct Dispatch
    : internal_::MapSpecExp<Exp<internal_::UndefinedFunSpec> >
    {};
```

Its parameters explicitly reflect information on the type category of the arguments as provided by their traits, some of which are expressed collectively to describe common properties of all arguments:

- `LEVEL_` is the maximal level of all arguments besides the 0th.
- `IS_DEFINED_` tells whether all arguments are defined.
- `DELAY_` tells whether the evaluation is to be delayed, whose precise interpretation will be discussed in § 9.3.1, until when this parameter should be assumed to be bound with `false`.
- `Stripped`*`N`*`_` is the bare (non-referenced, non-qualified) type of the *`N`*th argument.

An instance of `Dispatch` is a map specification meta expression of the mapping definition to be evaluated. This means that it contains the template `Result::Apply` which actually will be bound with the argument types, so that these are available within the core definition as well. By default, if no specialization matches (e.g., if any of the arguments is `Undefined`), the result of `Dispatch` and therefore of the mapping's binding is `Undefined`.

On instantiation, `Bind` binds `Dispatch` with its analyzed arguments and evaluates the provided `Result`. This process might not be intelligible at this point, as the meaning of the single properties has not been illustrated so far, but it is presented here to follow the order implied by functionality. The reader might skip the details first and eventually get back to them after having understood the implications.

⟨`Bind`⟩≡
    ⟨*auxiliary macros for definition of* `Bind`⟩
    ⟨*definition of* `Bind`⟩
    ⟨⟨*undefine auxiliary macros for definition of* `Bind`⟩⟩

To abbreviate the notation, the analyzation is wrapped into macros.

⟨*definition of* `Bind`⟩≡
```
template <class Exp0_, class Exp1_, class Exp2_, class Exp3_, class Exp4_>
struct Bind
: Dispatch<MaxLevel__,!(From0__(||,LEVEL,UNDEFINED)),
           !(Is__(0,EFFECT,IMMEDIATE)) && From1__(||,BINDING,DELAYED),
           Arg__(0),Arg__(1),Arg__(2),Arg__(3),Arg__(4)>::Result
  ::template Apply<Param__(0),Param__(1),Param__(2),Param__(3),Param__(4)>
{};
```

This reads as follows:

1. Obtain the maximum of the levels of arguments $1, \ldots,$ `N`
2. Starting from the 0th argument, check for each whether its `LEVEL` property is `UNDEFINED` and reduce the single results by the operator `||` to check whether this holds for any. Negate the result to indicate whether all arguments are defined.
3. Check whether the `EFFECT` of the 0th (the mapping) argument is not `IMMEDIATE` and whether the `BINDING` of any of the other arguments is `DELAYED`.
4. Strip each argument type.
5. Instantiate `Dispatch` and apply its `Result` to the argument types.

Though their implementation might seem obvious or a mere technical detail to some readers, the definition of these macros is given here in order to fully exhibit these central constructs to the curious.

⟨*auxiliary macros for definition of* `Bind`⟩≡
```
#define Param__(N__)                                                  \
    typename Exp##N__##_::Result

#define Traits__(N__)                                                 \
    internal_::Traits<Param__(N__)>

#define Is__(N__,PROPERTY__,VALUE__)                                  \
    ((int)(Traits__(N__)::PROPERTY__)==(int)(VALUE__))

#define From1__(OP__,PROPERTY__,VALUE__)                              \
    (     Is__(1,PROPERTY__,VALUE__)                                  \
     OP__ Is__(2,PROPERTY__,VALUE__)                                  \
     OP__ Is__(3,PROPERTY__,VALUE__)                                  \
     OP__ Is__(4,PROPERTY__,VALUE__))

#define From0__(OP__,PROPERTY__,VALUE__)                              \
    (     Is__(0,PROPERTY__,VALUE__)                                  \
     OP__ From1__(OP__,PROPERTY__,VALUE__))

#define Scope__(N) (int)(Traits__(N)::LEVEL)

#define Max__(A,B) (A)>(B) ? (A) : (B)

#define MaxLevel__                                               \
    (Level)(Max__(Max__(Max__(Scope__(1),Scope__(2)),Scope__(3)),Scope__(4)))

#define Arg__(N__)                                               \
    typename Traits__(N__)::ToStripped
```

***Remark 7.28***
*Thanks to the non-intrusive analyzation using traits, the dispatch of* `Bind` *does not require its arguments' types to be complete. Therefore, types do not even have to be defined in order to allow the evaluation of a mapping's binding at the meta stage.*

To abbreviate the notation of the presented constructs and to abstract their implementation from the number of arguments being limited to 4, symbolic replacements are used for the padding of parameter and argument lists by default parameters and arguments. `Fill`*`N`*`__` provides a comma-separated (and comma-initiated) list of as many instances of its argument as necessary to fill a list of *`N`* parameters or arguments to its full length.

⟨`Fill`*`N`*`__`⟩≡
```
    #define Fill4__(Item__)
    #define Fill3__(Item__) Fill4__(Item__),Item__
    #define Fill2__(Item__) Fill3__(Item__),Item__
    #define Fill1__(Item__) Fill2__(Item__),Item__
    #define Fill0__(Item__) Fill1__(Item__),Item__
```

# Chapter 8

# Naming

```
⟨implementation⟩+≡
    ⟨aliased operators⟩
    ⟨Named⟩
    namespace internal_
    {
        ⟨traits for internal named fun type⟩
    }
    ⟨named fun calls⟩
    ⟨global function definitions⟩
    ⟨bind function⟩
```

The common way of referring to a mapping is by an *identifier*—if it is a *named mapping*. This is the familiar case, but mappings may be and frequently are defined as *anonymous mappings*. There is a limited and fixed set of mapping identifiers, each of them referring to a mapping of explicitly defined semantics. This implies that such a mapping does not have a state and will therefore be defined as a *named fun*.

## 8.1 Identifiers

The definitions of both the identifiers and of specializations of the associated entities follow repeated schemes, which motivates to automatize them by means of macros to simplify this process and to avoid replication of hand-written code. This requires a thorough classification of mapping identifiers.

### 8.1.1 Classification

While code-producing macros will have to be re-defined according to their actual purpose, the set of mapping identifiers and their classification is invariant and expressed by a fixed set of macro calls of the following format, whose parameters are explained below:

```
    Named__(EFFECT,ARITY,Identifier,STAGE,identifier,DOMAIN,,,,,)
```

These calls are gathered within the file `named.hh`, which initially defines all undefined macros to be empty and finally undefines all macros. Hence, by including the macro call file `named.hh` after defining (some of) the macros, their contents are provided for each named mapping. This will be illustrated by oncoming applications.

The macro parameters used to identify or configure the appropriate identifier-dependent code definition describe the mapping as follows:

- *`EFFECT`* is the `Effect` value of the mapping.
- *`ARITY`* is the mapping's arity, a positive integer constant.
- *`Identifier`* (with uppercase initial) is the identifier of the mapping at the meta stage, which maps *`ARITY`* argument types to a member type `Result` and, if the mapping has soma stage semantics, to a static member function `result` that maps *`ARITY`* argument objects of the argument types to a `Result` object.
- *`STAGE`* is the lowest `Stage` to which the mapping's semantics apply.
- *`identifier`* (all lowercase) is the soma stage identifier of the mapping, which maps *`ARITY`* argument objects to an object of the `Result` type of the meta stage evaluation of the mapping bound with the arguments' types, if defined.
- *`DOMAIN`* is `DELAYED`, if the mapping is only defined for delayed mapping arguments. This holds for mappings whose semantics describe a certain order of evaluating their arguments, see § 9. Otherwise, *`DOMAIN`* is `ANY`.

### 8.1.2 Operators

Important special cases of named mappings are those representing operators, which may be overloaded in C++ to be equivalent to the mapping's soma function *`identifier`*. This causes difficulties in some cases, though, which will have to be treated specially:

- The *member access operator* `.` , [ISO98] § 5.2.5, the *pointer to member operator* `.*`, [ISO98] § 5.5, and the *scope resolution operator* `::` , [ISO98] § 5.1;7, can not be overloaded, [ISO98] § 13.5;3.
- The ternary *conditional operator* `? :` , [ISO98] § 5.16, can not be overloaded either, [ISO98] § 13.5;3. Its functionality will be provided, though, by the mapping `Cond` in § 9.4.2. Due to the lack of overloading, `cond` is not equivalent to, but more powerful than this operator.

  Note that the behaviour of `? :` can not be fully simulated rightaway: The order of evaluation of a function's arguments is unspecified, while all arguments are evaluated before the function is bound with them, [ISO98] § 5.2.2;8. This is in contrast with the conditional operator `? :` evaluating either just the second or just the third argument, depending on the value of the first.
- The binary logical operators `&&` , [ISO98] § 5.14, and `||` , [ISO98] § 5.15, may be overloaded, but as above their semantics requires to evaluate their second argument only if necessary, see § 9.4.2.
- The sequential operator `,` evaluates its arguments sequentially and returns the value of the last. As this holds for both builtin and user-defined arguments, it seems to not make sense to overload it, even more due to the undefined order of argument evaluation. But there are remarkable exceptions, see § 9.4.1.

- The *function call operator* `()` , [ISO98] § 5.2.2, on the one hand plays a special role on the border between types and functions and will therefore have to be defined and used differently from other operators. On the other hand, it may take an arbitrary number of parameters. Therefore, it will be defined explicitly where applicable.

- The pointer to member operator `->` , [ISO98] § 5.2.5, is a postfix operator, but its result (type) depends on both its argument type and on the type of the following class member identifier, [ISO98] § 5.2.5;1. This deduction can not be simulated by explicit constructs, wherefore it is not possible to embed this operator within the mapping framework presented here and the composition of the operators `*` and `.` have to be used instead. It is generally possible, though, to overload operator `->`—using prefix syntax—following the example given in [Ale01].

- For consistency, the binary *member pointer operator* `->*` , [ISO98] § 5.5, is not provided either, although its implementation would not cause the problems of operator `->` as in this case the result type can be derived from the second argument type.

Operators to overload automatically are specified by macro calls of the form

```
Named__(EFFECT,ARITY,Identifier,STAGE,identifier,DOMAIN,
        Position,OP_MODE,OP,OP_DOMAIN,Access)
```

which make use of the remaining five fields in the above specification of `Named__`:

- *`Position`* is the operator's position relative to its argument(s), with the following choices:

  – for unary operators:

    * `Prefix` or
    * `Postfix`

  – for binary operators:

    * `Infix` or
    * `Pair`

    `Pair` describes a *pair operator* *`LOP ROP`* consisting of two parts. As the function call operator `()` was excluded from automatic overloading above, this applies only to the subscript operator `[]` , [ISO98] § 5.2.1.

- *`OP_MODE`* usually is chosen as `DIRECT`, but see the next item.

- *`OP`* is the operator itself. Because operator `,` (like operator `()`) can not be supplied as a macro argument, and to provide operator `[]` as a single argument, these operators are not given explicitly, but the preceding parameter *`OP_MODE`* is then set to the identifiers `COMMA` or `BRACK`.

  To obtain the appropriate function identifier for a mapping, the identifier `Mode` will be concatenated with the value of *`OP_MODE`* and the resulting macro is applied to the automatically created function identifier, which is usually preserved, but replaced in the mentioned cases.

⟨*aliased operators*⟩≡
```
    #define ModeDIRECT__(id__)                                          \
    id__

    #define ModeCOMMA__(id__)                                           \
    operator,

    #define ModeBRACK__(id__)                                           \
    operator[]
```

As operator `()` will be implemented separately, no according ***`OP_MODE`*** value `PAREN` is introduced.

- ***`OP_DOMAIN`*** is the domain of the operator version of the mapping, which is typically but not always equal to the ***`Domain`*** of the named version, see § 11.1.3.
- ***`Access`*** indicates where to define an overloading function and which access the operator requires to its arguments, if they are of builtin type:
  - `Class` marks operators that have to be overloaded by member functions. This is required by the C++ language to ensure that their first parameter is an lvalue. This is the case for the assignment operator `=` , the (pointer to) class member access operator `->` and the paired operators `()` and `[]` , [ISO98] § 13.5.3-6, of which only the first and the last have to be considered according to the above considerations.
  - `Lvalue` marks operators that require builtin type arguments to be lvalues as well, but which may be overloaded by a global function. This will prove more convenient to implement than overloading by a member function.
  - `Rvalue` marks those operators which are applicable to builtin type rvalue arguments, thus to compile-time constants as well.

Therefore, of all combinations of ***`Access`*** and ***`Position`*** only the ones given in table 8.1 remain to be considered.

| | `Prefix` | `Postfix` | `Infix` | `Pair` |
|---|---|---|---|---|
| `Rvalue` | ● | | ● | |
| `Lvalue` | ● | ● | ● | |
| `Class` | | | ● | ● |

Table 8.1: ***`Access`*** and ***`Position`*** combinations

Most mappings associated with operators have `INDIVIDUAL` effect and `ANY` domain, except:

- prefix operator `&` having `GENERAL` effect and `DELAYED` domain, see § 11.1.3.
- operator `,` and operators `&&` and `||` having `IMMEDIATE` effect and `DELAYED` domain, see § 9.4.1 and § 9.4.2.

## 8.2 Definitions

### 8.2.1 Funs

The definition of named mappings using the presented macros will be illustrated by the example of binary named mappings; mappings of other arities are defined equivalently. As mentioned, a named mapping in fact is a named fun, whose specification is an instance of the template class `Named`.

⟨`Named`⟩≡
```
    template <Effect EFFECT_, natural ARITY_, class Tag_>
    struct Named
    {};
```

Except for `Tag_`, its parameters should be self-explanatory. `Tag_` is to be bound with the essential representation of a named mapping, given as an undefined type called *`Identifier`*`Tag`, which has no further meaning than to be identifiable.

Obviously, a named mapping carries its type category within its type identifier, which can be extracted directly by a traits specialization.

⟨*traits for internal named fun type*⟩≡
```
    template <typename Type_,
              Binding BINDING_, Effect EFFECT_, natural ARITY_, class Tag_,
              typename Pointer_>
    struct UnqualifiedDispatch_<Type_,
                                Internal<META,BINDING_,Named<EFFECT_,ARITY_,Tag_> >,
                                Pointer_>
    : ClassTraits<Type_>
    {
        enum
        {
            LEVEL = INTERNAL_META,
            BINDING = BINDING_,
            EFFECT = EFFECT_
        };
    };
```

### 8.2.2 Calls

⟨*named fun calls*⟩≡
```
    ⟨internal_::FunBind_⟩
    ⟨⟨1ary named fun calls⟩⟩
    ⟨2ary named fun calls⟩
    ⟨⟨3ary named fun calls⟩⟩
    ⟨⟨4ary named fun calls⟩⟩
    ⟨Named__ specialization for named fun call⟩
    #include "named.hh"
    ⟨⟨undefine macros for named fun call⟩⟩
```

Now everything is set to define the processing of named fun calls. As mentioned in § 7.4.1, it should be possible to denote such a call like

```
    Identifier<Exp_1,...,Exp_ARITY>
```

whose instances should obtain their contents from the according instance of `Bind`, passing it the mapping itself as 0th argument to abstract from the fact that the mapping is a fun. This step is wrapped within the auxiliary template class `FunBind_`.

⟨`internal_::FunBind_`⟩≡
```
namespace internal_
{
    template <class Exp0_,
              class Exp1_=Exp<Void>, class Exp2_=Exp<Void>,
              class Exp3_=Exp<Void>, class Exp4_=Exp<Void> >
    struct FunBind_
    {
    private:
        typedef Bind<Exp0_,Exp1_,Exp2_,Exp3_,Exp4_> Bind_;
    public:
        typedef typename Bind_::Result Result;
        static Result result(typename Exp1_::Result param1,
                             typename Exp2_::Result param2,
                             typename Exp3_::Result param3,
                             typename Exp4_::Result param4)
        {
            return Bind_::result(typename Exp0_::Result(),
                                 param1,param2,param3,param4);
        }
    };
}
```
A fun call just instanciates and inherits an instance of `FunBind_`.

⟨*2ary named fun calls*⟩≡
```
#define Arity2__(Id__,EFFECT__)                                         \
template <class Exp1_, class Exp2_>                                     \
struct Id__                                                             \
: internal_::FunBind_                                                   \
<Exp<Internal<META,ACTUAL,Named<EFFECT__,2ul,Id__##Tag> > >,            \
 Exp1_,Exp2_                                                            \
>                                                                       \
{};
```
The general macro defines the identifying tag and invokes the call definition of appropriate arity.

⟨`Named__` *specialization for named fun call*⟩≡
```
#define Named__(EFFECT__,ARITY__,Id__,STAGE__,id__,DOMAIN__,            \
                Position__,OP_MODE__,OP__,OP_DOMAIN__,Access__)         \
struct Id__##Tag;                                                       \
Arity##ARITY__##__(Id__,EFFECT__)
```
Within file `named.hh` included hereafter, this macro is called for each named mapping and all those definitions are generated. This rather simple example should illustrate the automatization, particularly the dispatch technique by concatenating the name of a property with its actual value, as is done here for the mapping's arity. This method of generating named definitions will be applied in several occasions like the following.

### 8.2.3 Global functions

After the definition of named fun calls to be evaluated at the meta stage, named functions have to be defined for soma stage computations. While the former just have to consider type expressions, which are passed without any transformation as template arguments, the latter may be bound with object arguments of different type, that have to be appropriately detected using the overload resolution rules. Another complicating issue are the subtleties of C++ function definitions which have to be respected, including those of operator overloading.

According to solution 7.6, soma stage mapping evaluation should be expressed by calls of generic global functions, which accept arguments of any type. For operators and builtin type arguments, though, the builtin definition is automatically considered, which avoids both the necessity of providing appropriate definitions and the problem of recursive definitions: for instance, the explicitly defined addition of two `int` values would have to be expressed by itself. Still, the appropriate return type has to be provided, which will be arranged in § 10.3.1.

Another argument property to detect is its cv-qualification. While a mapping's semantics should not depend on the qualification of its arguments—wherefore this is not considered part of its type category—its implementation and its return type may. How is an argument's qualification preserved and detected by a generic function call? The argument may be given as an rvalue or as an lvalue, while the parameter type may be an object type or a reference type.

***Problem 8.1***
*These alternatives do not match each other, though: If the function is overloaded to accept either any type or a reference type, neither the call by an rvalue is uniquely matched with the former nor that by an lvalue with the latter, [ISO98] § 14.8.2.1, wherefore this coexistence of both variants can not be supported.*

The unconstrained generic variant accepting any parameter type causes any argument to be passed by value. This means that it can not be accessed directly from within the function and that unnecessary costs arise if the argument is expensive to copy. Consequently, the argument's qualification is discarded, which can not be influenced by overloading the function qualifying the parameter.

A generic function accepting reference parameters behaves just conversely: Arguments are passed by reference, which requires the creation of a *temporary* `const`-qualified lvalue for an rvalue argument according to [ISO98] § 8.5.3, even if it is of a scalar or some other type cheap to copy. This conversion is only enforced, though, if an overloaded version of the function exists that takes a generic `const`-qualified reference parameter.

***Remark 8.2***
*It is sufficient to overload a generic function by* `const` *and non-*`const` *reference parameter types to respect the* `volatile`*-qualified cases as well, [ISO98] § 12.8;2.*

***Solution 8.3 (to problem 8.1)***
*Consequently, a generic function should have reference parameters and be overloaded to provide all combinations of no qualification and* `const`*-qualification for each of its parameters. Unnessecary costs arising from the creation of temporaries may be eliminated by the compiler if these are only used as rvalues within the function.*

***Suggestion 8.4***
*The proposals [HDA02, HAD04, AAD$^+$05], combined with [Koz05], would allow to explicitly overload functions to accept only rvalue or only lvalue parameters to avoid the creation of temporaries. This would solve this remaining problem, but it would also allow to define identifiers to denote builtin type rvalues, which can only be simulated so far by defining* `enum` *values, [VJ03]* § *17.2.*

***Remark 8.5***
*The consequent use of reference parameters also allows to pass an argument's non-converted and non-promoted type, e.g., a* `char` *value as such instead of passing it as an* `int` *value.*

Besides the binary case, function implementations which are particularly interesting to illustrate are unary ones, which also have to support operator overloading, requiring special consideration of the case of a postfix operator.

⟨*global function definitions*⟩≡
```
    ⟨auxiliary macros for global function definitions⟩
    ⟨1ary global function definitions⟩
    ⟨2ary global function definitions⟩
    ⟨⟨3ary global function definitions⟩⟩
    ⟨⟨4ary global function definitions⟩⟩
    ⟨Named__ specialization for global function definitions⟩
    #include "named.hh"
    ⟨⟨undefine macros for global function definitions⟩⟩
```

First, some auxiliary macros are defined to encapsulate details of the function definitions. To enforce an argument to have delayed binding, as necessary for a mapping of `DELAYED Domain`, the `INDEX__`th formal and actual template parameter types are given by the macro of the name resulting from the concatenation of both terms. If applicable, the argument is forced to be a `DELAYED` instance of `Internal`.

⟨*auxiliary macros for global function definitions*⟩≡
```
    #define FormalANY__(INDEX__)                                     \
    typename Stripped##INDEX__##_

    #define ActualANY__(INDEX__)                                     \
    Stripped##INDEX__##_

    #define FormalDELAYED__(INDEX__)                                 \
    Stage STAGE##INDEX__##_, class Mapping##INDEX__##_

    #define ActualDELAYED__(INDEX__)                                 \
    Internal<STAGE##INDEX__##_,DELAYED,Mapping##INDEX__##_>
```

Another construct provides an eventual dummy `int` parameter type to mark postfix operator definitions.

⟨*auxiliary macros for global function definitions*⟩+≡
```
    #define Position__
    #define PositionPrefix__
    #define PositionPostfix__                                        \
    , int
    #define PositionInfix__
    #define PositionPair__
```

The definition of an unary function uses the previous constructs to deduce the formal and actual parameter type and to supply an eventual postfix operator marker by concatenating the according prefix with the according property identifier argument. The `Mode...` construct introduced in § 8.1.2 replaces the provided function identifier with its correct representation. In fact, this is unnecessary for the unary case, as only the binary operators `,` and `[]` are aliased, but it is used here for consistency of notation.

The function is `inlined`, as it only consists of a forwarding call of the `result` member of the appropriate fun call, which also provides the return type.

⟨*1ary global function definitions*⟩≡
```
    #define Arity1Qual1__(Id__,Position__,MODE__,id__,DOMAIN__,QUAL1__)          \
    template <Formal##DOMAIN__##__(1)>                                             \
    inline                                                                         \
    typename Id__<Exp<Actual##DOMAIN__##__(1) QUAL1__&> >::Result                  \
    Mode##MODE__##__(id__) (Actual##DOMAIN__##__(1) QUAL1__& param1                \
                            Position##Position__##__)                              \
    {                                                                              \
        return Id__<Exp<Actual##DOMAIN__##__(1) QUAL1__&> >                        \
               ::result(param1 Fill1__(_));                                        \
    }
```
This definition has to be provided twice, to let the reference parameter be `const`-qualified or not.

⟨*1ary global function definitions*⟩+≡
```
    #define Arity1__(Id__,Position__,MODE__,id__,DOMAIN__)                         \
    Arity1Qual1__(Id__,Position__,MODE__,id__,DOMAIN__,const)                      \
    Arity1Qual1__(Id__,Position__,MODE__,id__,DOMAIN__,     )
```
That's it for the definition of any named unary function! The macro `Arity1__` remains to be called, but there is no more code to be provided on the meta or soma stage. The implementor will not have to be aware of the syntax of a function definition and the necessary overloads anymore, but just has to provide an appropriate declaration of the mapping by a `Named__` call and according fun specification definitions which obey to a much stricter layout, as will be demonstrated later on.

***Remark 8.6***
*This application of the automatization of named mapping definitions suggests to consider the preprocessing stage another stage entered before the meta and the soma stage, which refer to the compilation and the execution stage of the program. While macros are not considered good programming practice if used to replace constructs of other stages, like constants or functions, their application to generate code by operations on literals (the objects of this stage) is a pure and self-contained process. The* Boost Preprocessor Library *[MK, AG05] contains preprocessing stage algorithms and even data structures, though not mixing its syntax with that of other stages. It should be interesting to combine it with the techniques presented here to a three-stage approach that allows to write code that is generic even on the literal level. One very relevant effect would be the abstraction from the particular choice of a maximal parameter number.*

***Remark 8.7***
*As the meta stage code now is subject to transformations just like soma stage code was before, some terms have to be qualified:*

- *The* absolute stage *is an element of the set*

  $$\{\ldots, preprocessing, compilation, execution, \ldots\}$$

  *which is ordered in the indicated way. An element located to the right of another is said to be* later*, if to the left, it is* earlier*. The given stages are* adjacent*, i.e., there is no stage between them.*

  *This allows to view the set as a sequence and to map it isomorphically to integer numbers, preserving the order by mapping earlier stages to higher numbers. Assuming the execution stage to be mapped to 0, positive numbers will refer to stages dealing with program generation (before the mentioned: configuration, tangling, parsing), negative numbers to its evaluation (machine instructions, memory states, in- and output). These ideas are related to* multistage programming *[Tahb] as expressed in the language* MetaOCamL *[Taha].*

- *The classification as a* relative stage *depends on the context. If two (and only two) adjacent stages participate in a process, the earlier of them is called the meta stage and the later the soma stage.*

*In the absence of a particularly mentioned context, though, the conventional usage of the terms* meta *and* soma stage *shall be kept.*

The definition of a binary function is even simpler than that of an unary one, as no postfix marker has to be considered. Now, though, it is essential to use the `Mode...` macros to obtain the correct function identifier.

⟨*2ary global function definitions*⟩≡
```
#define Arity2Qual2__(Id__,MODE__,id__,DOMAIN__,QUAL1__,QUAL2__)          \
template <Formal##DOMAIN__##__(1),Formal##DOMAIN__##__(2)>               \
inline                                                                    \
typename Id__                                                             \
<Exp<Actual##DOMAIN__##__(1) QUAL1__&>,                                   \
 Exp<Actual##DOMAIN__##__(2) QUAL2__&>                                    \
>::Result                                                                 \
Mode##MODE__##__(id__) (Actual##DOMAIN__##__(1) QUAL1__& param1,          \
                        Actual##DOMAIN__##__(2) QUAL2__& param2)          \
{                                                                         \
    return Id__<Exp<Actual##DOMAIN__##__(1) QUAL1__&>,                    \
                Exp<Actual##DOMAIN__##__(2) QUAL2__&> >                   \
           ::result(param1,param2 Fill2__(_));                            \
}
```
To provide the necessary overloads, though, two steps have to be performed to qualify each of both parameters. The responsible preprocessing stage code increases linearly with the argument number, causing quadratic order of overall code size.

***Remark 8.8***
*By preprocessing stage algorithms, this effort can be made constant. Given a moderate maximal parameter number of 4, this improvement would not be relevant here, though.*

⟨*2ary global function definitions*⟩+≡
```
#define Arity2Qual1__(Id__,MODE__,id__,DOMAIN__,QUAL1__)                    \
Arity2Qual2__(Id__,MODE__,id__,DOMAIN__,QUAL1__,const)                      \
Arity2Qual2__(Id__,MODE__,id__,DOMAIN__,QUAL1__,     )

#define Arity2__(Id__,Position__,MODE__,id__,DOMAIN__)                      \
Arity2Qual1__(Id__,MODE__,id__,DOMAIN__,const)                              \
Arity2Qual1__(Id__,MODE__,id__,DOMAIN__,     )
```
Finally, the appropriate macro has to be called, based on the mapping's arity. For mappings associated with an operator that does not have to be overloaded by a member function, both the version named by an identifier and the one named `operator OP__` have to be provided.

⟨`Named__` *specialization for global function definitions*⟩≡
```
#define Access__(Id__,ARITY__,id__,DOMAIN__,                                \
                Position__,OP_MODE__,OP__,OP_DOMAIN__)                      \
Arity##ARITY__##__(Id__,,DIRECT,id__,DOMAIN__)

#define AccessRvalue__(Id__,ARITY__,id__,DOMAIN__,                          \
                      Position__,OP_MODE__,OP__,OP_DOMAIN__)                \
Arity##ARITY__##__(Id__,,DIRECT,id__,DOMAIN__)                              \
Arity##ARITY__##__(Id__,Position__,OP_MODE__,operator OP__,OP_DOMAIN__)

#define AccessLvalue__(Id__,ARITY__,id__,DOMAIN__,                          \
                      Position__,OP_MODE__,OP__,OP_DOMAIN__)                \
Arity##ARITY__##__(Id__,,DIRECT,id__,DOMAIN__)                              \
Arity##ARITY__##__(Id__,Position__,OP_MODE__,operator OP__,OP_DOMAIN__)

#define AccessClass__(Id__,ARITY__,id__,DOMAIN__,                           \
                     Position__,OP_MODE__,OP__,OP_DOMAIN__)                 \
Arity##ARITY__##__(Id__,,DIRECT,id__,DOMAIN__)
```
All this only has to be applied if the mapping has soma stage semantics, which is detected by the last dispatch called by `Named__`.

⟨`Named__` *specialization for global function definitions*⟩+≡
```
#define StageMETA__(Id__,ARITY__,id__,DOMAIN__,                             \
                   Position__,OP_MODE__,OP__,OP_DOMAIN__,Access__)

#define StageSOMA__(Id__,ARITY__,id__,DOMAIN__,                             \
                   Position__,OP_MODE__,OP__,OP_DOMAIN__,Access__)          \
Access##Access__##__(Id__,ARITY__,id__,DOMAIN__,                            \
                    Position__,OP_MODE__,OP__,OP_DOMAIN__)


#define Named__(EFFECT__,ARITY__,Id__,STAGE__,id__,DOMAIN__,                \
               Position__,OP_MODE__,OP__,OP_DOMAIN__,Access__)              \
Stage##STAGE__##__(Id__,ARITY__,id__,DOMAIN__,                              \
                  Position__,OP_MODE__,OP__,OP_DOMAIN__,Access__)
```

### 8.2.4 Anonymous function call

Besides those named functions, a global function `bind` allowing an *anonymous function* call analogous to the anonymous mapping evaluation by `Bind` is defined.

⟨bind *function*⟩≡
    ⟨⟨*4ary* bind *function definition*⟩⟩
    ⟨⟨*3ary* bind *function definition*⟩⟩
    ⟨*2ary* bind *function definition*⟩
    ⟨⟨*1ary* bind *function definition*⟩⟩
    ⟨⟨*0ary* bind *function definition*⟩⟩
    ⟨⟨bind *function definition*⟩⟩
    ⟨⟨*undefine macros for* bind *function definition*⟩⟩

The definition follows very much the structure introduced in § 8.2.3, but it takes a 0th parameter and can therefore not be integrated smoothly there. Furthermore, it has to be overloaded for any arity, as the number of arguments may be arbitrary in this case.

⟨*2ary* bind *function definition*⟩≡
```
#define BindQual2__(QUAL0__,QUAL1__,QUAL2__)                        \
BindQual3__(QUAL0__,QUAL1__,QUAL2__,const)                          \
BindQual3__(QUAL0__,QUAL1__,QUAL2__,     )                          \
template <typename Stripped0_,                                      \
          typename Stripped1_, typename Stripped2_>                 \
inline                                                              \
typename Bind<Exp<Stripped0_ QUAL0__&>,                             \
              Exp<Stripped1_ QUAL1__&>,Exp<Stripped2_ QUAL2__&> >   \
         ::Result                                                   \
bind(Stripped0_ QUAL0__& param0,                                    \
     Stripped1_ QUAL1__& param1, Stripped2_ QUAL2__& param2)        \
{                                                                   \
    return Bind<Exp<Stripped0_ QUAL0__&>,                           \
                Exp<Stripped1_ QUAL1__&>,Exp<Stripped2_ QUAL2__&> > \
           ::result(param0,param1,param2 Fill2__(_));               \
}
```

Instead of global functions, stateless global fun type objects providing a function call operator `()` could be used as well to implement named soma mappings. This approach is actually taken in the FC++ library [Sma], but was not followed here for different reasons:

- The introduction of global objects is on the one hand considered "impure" in the common sense of different programming paradigms, on the other hand requires separate compilation of run-time code to use them and appropriate linking to avoid ambiguities. Both does not coincide with the idea of a "language extension" which should be lightweight, without causing code size or application complexity overhead.

- The introduction of functions as first-class lvalues bears the temptation to use them as such, which contradicts the concept of a function being unique and invariant. One would have to disallow to modify it (which could be prevented by `const`-qualification, but then again be circumvented by a malicious cast), to create further instances of it (by making it a singleton, which is non-trivial) or to apply qualifiers to it. So one would have to invert many efforts to make such an object behave—just like a global function. Those cases in which object behaviour of a function is desired, e.g., to pass it to a higher-order mapping, can be achieved in a different way.

- While overloading the function call operator `()` for a fun type object would allow to use either notation of `*fun*(...)` or `bind(*fun*,...)`, there is no analogon at the meta stage. ***`Fun`*** would have to be a template and could therefore not be argument to `Bind`. Thus, no convenient way is provided to pass a global fun object as 0th argument to `bind` with intention. And even if someone constructed an according object by hand to do so, it would have meta semantics and not cause the creation of any additional soma code.

- A function should merely provide syntactic sugar to comfortably initiate the appropriate mapping evaluation process. As little information as possible should be individualized, in particular should all technical matters be encapsulated within `Bind` and its associated constructs. If the definition of the global fun type contains nothing more than the according function definition, though, it makes no sense to wrap that into a class.

# Part III

# Applications

# Chapter 9

# Evaluation

⟨*implementation*⟩+≡
    ⟨*default evaluation*⟩
    ⟨*immediate evaluation*⟩
    ⟨*delayed evaluation*⟩
    ⟨*non-parallel evaluation*⟩

The template class `Dispatch` may be (partially) specialized in many ways to provide different behaviour on its instantiation. Besides the semantics of the particular mapping that is to be evaluated, there are some more general principles of evaluation which are reflected by common patterns of specialization and implementation of the according instances. The mappings presented here therefore have some higher-order semantics which directly relates to those principles.

## 9.1 Default evaluation

⟨*default evaluation*⟩≡
    ⟨*default map evaluation*⟩
    ⟨*default fun call*⟩

The discussion shall be started by considering the absence of peculiarities: As long as the parameter `DELAY_` of a `Dispatch` specialization is bound with `false`, the definition of a mapping by default is given by itself.

⟨*default map evaluation*⟩≡
```
    template <Level LEVEL_, Binding BINDING0_, class MapSpec0_,
              typename Stripped1_, typename Stripped2_,
              typename Stripped3_, typename Stripped4_>
    struct Dispatch<LEVEL_,true,false,
                    Internal<SOMA,BINDING0_,MapSpec0_>,
                    Stripped1_,Stripped2_,Stripped3_,Stripped4_>
    : Exp<MapSpec0_>
    {};
```

As motivated in § 7.3.2, this has to be specialized for arguments of fun type to avoid that these need to be generally given by mapping specifications.

⟨*default fun call*⟩≡
```
    template<Level LEVEL_, Binding BINDING0_, class FunSpec0_,
          typename Stripped1_, typename Stripped2_,
          typename Stripped3_, typename Stripped4_>
struct Dispatch<LEVEL_,true,false,
               Internal<META,BINDING0_,FunSpec0_>,
               Stripped1_,Stripped2_,Stripped3_,Stripped4_>
: internal_::MapSpecExp<Exp<FunSpec0_> >
{};
```

Therefore, the definition of a mapping may be provided without further knowledge of the evaluation process, as long as it obeys the standard form of a fun or map specification. The definition of a mapping by a specialization of `Dispatch` is only necessary for those cases where the implementation depends on the arguments'

- type category or
- (stripped) exact or partially specialized type.

## 9.2 Immediate evaluation

⟨*immediate evaluation*⟩≡
```
    ⟨Make⟩
    ⟨ToNonDelayed⟩
    ⟨ToDelayed⟩
```

As motivated in § 5.2.5, delayed evaluation is superseded by immediate mappings. These provide important functionalites to deal appropriately with delayed arguments circumventing the general way of automatized creation of a new delayed mapping. As a natural consequence, any immediate mapping object has `ACTUAL` binding.

### 9.2.1 Object construction

⟨`Make`⟩≡
```
    namespace internal_
    {
        ⟨DefaultConstructFunSpec_⟩
        ⟨make⟩
        ⟨⟨ConstructFunSpec_⟩⟩
        ⟨MakeFunSpec_⟩
        ⟨MakeFunSpec_ with meta type argument⟩
    }
    ⟨Dispatch for Make⟩
```

The most essential mapping is the immediate meta semantics fun `Make`

⟨*named fun classifications*⟩≡
```
    Named__(IMMEDIATE,1,Make,META,make,ANY,,,,,)
```

which returns a delayed soma semantics fun that creates an object of the argument type. The result has no particular arity, as there may be different constructors of different arities for that type. This has no particular effect for a meta semantics argument, which in any case is just default-constructed. Therefore, an auxiliary fun specification is specialized to dispatch this information.

⟨DefaultConstructFunSpec_⟩≡
```
template <typename Result_>
struct DefaultConstructFunSpec_
{
    template <typename Param1_, typename Param2_,
              typename Param3_, typename Param4_>
    struct Call
    {
        typedef Result_ Result;
        static Result result(Param1_, Param2_, Param3_, Param4_)
        {
            return Result();
        }
    };
};
```
If the argument has soma semantics, an overloaded function has to perform the appropriate constructor call for each number of (non-`Void`) arguments, of which only the binary case is outlined here.

⟨make⟩+≡
```
template <typename Result_,
          typename Param1_, typename Param2_>
static Result_ make(Param1_ param1, Param2_ param2, Void, Void)
{
    return Result_(param1,param2);
}
```
The fun specification for a soma semantics argument straightforwardly calls that function, which does not have to be demonstrated. The creation of the appropriate fun specification can be expressed by the construction fun specifications themselves.

⟨MakeFunSpec_⟩≡
```
template <typename Stripped1_>
struct MakeFunSpec_
{
    template <typename Param1_, typename Param2_,
              typename Param3_, typename Param4_>
    struct Call
    : internal_::DefaultConstructFunSpec_
      <Internal<META,DELAYED,internal_::ConstructFunSpec_<Param1_> >
      >::template Call<Param1_,Param2_,Param3_,Param4_>
    {};
};
```
⟨MakeFunSpec_ *with meta type argument*⟩≡
```
template <Binding BINDING1_, class Specification1_>
struct MakeFunSpec_<Internal<META,BINDING1_,Specification1_> >
: internal_::DefaultConstructFunSpec_
  <Internal<META,DELAYED,internal_::DefaultConstructFunSpec_
   <Internal<META,BINDING1_,Specification1_>
   > >
  >
{};
```

The Dispatch specialization is then implemented directly by this fun specification. This process seems rather inconvenient due to the multiple steps of dispatching, but from now on, the Construct mapping may be used to create objects in other mapping evaluation definitions, wherefore according resolutions will not have to be performed anymore.

```
⟨Dispatch for Make⟩≡
    template <Level LEVEL_, typename Stripped1_>
    struct Dispatch<LEVEL_,true,false,
                    Internal<META,ACTUAL,Named<IMMEDIATE,1ul,MakeTag> >,
                    Stripped1_
                    Fill1__(Void)>
    : internal_::MapSpecExp<Exp<internal_::MakeFunSpec_<Stripped1_> > >
    {};
```

***Remark 9.1***
*The representation of object construction by a mapping allows to integrate this process into the common mapping evaluation framework. In contrast, a constructor may not be passed to a higher-order mapping as argument.*

***Remark 9.2***
*This mapping provides another important effect: The argument type is given by a meta expression, which possibly is formed by some non-trivial meta mapping evaluation. Of this type—whichever it may be—an object is constructed, given appropriate constructor arguments.*

*This situation describes a major application of the suggested* typeof *operator, see problem 7.12, which in such a context can be replaced by using* Make*.*

## 9.2.2 Undelaying objects

```
⟨ToNonDelayed⟩≡
    namespace internal_
    {
        ⟨⟨IdentityFunSpec_⟩⟩
    }
    ⟨⟨ToNonDelayed with actual type argument⟩⟩
    ⟨ToNonDelayed with delayed meta type argument⟩
    ⟨⟨ToNonDelayed with delayed soma type argument⟩⟩
```

Another very basic operation which was motivated in § 5.2.5 is to erase the property of an object to be a delayed mapping evaluation. This is necessary to allow to pass it as a plain (higher-order) argument to another mapping, without enforcing that to be delayed. As this information is given by the Binding_ parameter of the template class Internal, the sole purpose of the mapping ToNonDelayed is to return an object of equivalent semantics as the argument object, but with this parameter bound to ACTUAL.

```
⟨named fun classifications⟩+≡
    Named__(IMMEDIATE,1,ToNonDelayed,META,toNonDelayed,ANY,,,,,)
```

If the argument is not delayed, itself can be returned. For this purpose, a fun specification IdentityFunSpec_ is defined, which just returns its first argument. Its definition is bare of astonishing details, as is that of the according Dispatch, wherefore neither is shown here.

If the argument is delayed and of meta semantics, a new object of `ACTUAL` binding is created and returned.

⟨`ToNonDelayed` *with delayed meta type argument*⟩≡
```
    template <class Fun1_>
    struct Dispatch<INTERNAL_META,true,false,
                    Internal<META,ACTUAL,Named<IMMEDIATE,1ul,ToNonDelayedTag> >,
                    Internal<META,DELAYED,Fun1_>
                    Fill1__(Void)>
    : internal_::MapSpecExp<Make<Exp<Internal<META,ACTUAL,Fun1_> > > >
    {};
```

### 9.2.3 Delaying an object

⟨`ToDelayed`⟩≡
```
    ⟨⟨ToDelayed with delayed type argument⟩⟩
    ⟨ToDelayed with actual meta type argument⟩
    ⟨⟨ToDelayed with actual soma type argument⟩⟩
```

Symmetric to the previous case is the question of how to deal with actual arguments participating in an expression whose evaluation is to be delayed. Their state has to be preserved until the evaluation actually takes place, which can be achieved by binding such an argument within a delayed mapping that returns the argument when evaluated. This transformation is performed by the mapping `ToDelayed`, which is evaluated immediately in order to behave as the identity for a delayed argument.

⟨*named fun classifications*⟩+≡
```
    Named__(IMMEDIATE,1,ToDelayed,SOMA,toDelayed,ANY,,,,,)
```

Besides mentioning the use of the fun specification `IdentityFunSpec_` for this purpose, nothing particular has to be said about the implementation of that case. For an actual argument, the described binding has to be performed. Given a meta semantics argument, there is no state to be preserved, but only an object of the according type to be constructed when evaluating the delayed mapping. This can be implemented by means of the mapping `Make` and its according fun specification for meta objects introduced in § 9.2.1.

⟨`ToDelayed` *with actual meta type argument*⟩≡
```
    template <Level LEVEL_, class Specification1_>
    struct Dispatch<LEVEL_,true,false,
                    Internal<META,ACTUAL,Named<IMMEDIATE,1ul,ToDelayedTag> >,
                    Internal<META,ACTUAL,Specification1_>
                    Fill1__(Void)>
    : internal_::MapSpecExp<Make<Make<Exp
      <Internal<META,ACTUAL,Specification1_>
      > > > >
    {};
```

Note that a delayable mapping argument is just considered to be an object (in fact, there is no explicit information to distinguish a mapping from a non-mapping object), wherefore it is bound to be returned on demand, instead of being transformed to its delayed counterpart.

## 9.3 Delayed evaluation

⟨*delayed evaluation*⟩≡
    ⟨*delayable mapping with delayed and actual arguments*⟩
    ⟨*delayable mapping with delayed arguments*⟩

The immediate mappings introduced in the previous provide the necessary functionalities to discuss now the way to automatically delay the evaluation of a mapping. According to § 5.2.5, this is initiated whenever a delayable mapping is evaluated for delayed arguments.

### 9.3.1 Presence of actual arguments

If there are actual arguments besides the delayed ones, these are transformed to delayed mapping evaluations in order to allow a generalized implementation of the delayed evaluation itself. The mapping `ToDelayed` is suited for this task, as it does not modify delayed objects, so that it can be applied to all arguments and no particular specialization is needed to account for different combinations of delayed and actual arguments.

Nothing else is provided by this `Dispatch` specialization which only takes care of providing all arguments as delayed to the inner `Bind` instance, whose evaluation causing the creation of the delayed expression itself remains to be provided and discussed separately.

⟨*delayable mapping with delayed and actual arguments*⟩≡
```
template<Level LEVEL_, typename Stripped0_,
          typename Stripped1_, typename Stripped2_,
          typename Stripped3_, typename Stripped4_>
struct Dispatch<LEVEL_,true,true,
                Stripped0_,Stripped1_,Stripped2_,Stripped3_,Stripped4_>
{
    struct Result
    {
        template <typename Param0_,
                  typename Param1_, typename Param2_,
                  typename Param3_, typename Param4_>
        struct Apply
        {
        private:
            typedef ToDelayed<Exp<Param1_> > Delayed1_;
            typedef ToDelayed<Exp<Param2_> > Delayed2_;
            typedef ToDelayed<Exp<Param3_> > Delayed3_;
            typedef ToDelayed<Exp<Param4_> > Delayed4_;
            typedef Bind<Exp<Param0_>,
                         Delayed1_,Delayed2_,Delayed3_,Delayed4_> Bind_;
        public:
            typedef typename Bind_::Result Result;
            static Result result(Param0_ param0,
                                 Param1_ param1, Param2_ param2,
                                 Param3_ param3, Param4_ param4)
            {
                return Bind_::result(param0,
                                     Delayed1_::result(param1 Fill1__(_)),
                                     Delayed2_::result(param2 Fill1__(_)),
                                     Delayed3_::result(param3 Fill1__(_)),
                                        Delayed4_::result(param4 Fill1__(_)));
                }
            };
        };
    };
```

To implement this solution, the parameter `Delay_` of `Dispatch` had to be introduced in order to indicate whether any argument of a delayable (non-immediate) mapping is delayed and the evaluation to be delayed, without the need to inspect the single arguments whether they are delayed or not.

***Remark 9.3***
*Note that no special consideration was given to the arity of the mapping. Eventual default arguments of type* `Void` *are transformed into delayed mappings just like any other bound argument and lead again to the provision of default arguments in the moment of their evaluation. As all this happens at the meta stage, it should not cause any overhead while proving comfortable to implement.*

***Remark 9.4***
*While this approach provides a generic way to deal with partially bound expressions by "unbinding" the specified arguments, one may overwrite it by specializations that consider certain combinations of delayed and actual arguments or of type categories or types of the latter. By doing so, one may specify how to partially evaluate the given mapping. Such will also result in a delayed mapping, but in an adapted one which performs computations depending on actual arguments at the time of its creation rather than the time of its evaluation. This way, powerful optimizations by partial evaluation may be introduced separately and in a straightforward manner.*

### 9.3.2 Absence of actual arguments

⟨*delayable mapping with delayed arguments*⟩≡
```
    namespace internal_
    {
        ⟨ComposeFunSpec_⟩
    }
    ⟨delayable fun with delayed meta arguments⟩
    ⟨⟨delayable mapping with delayed arguments, any of soma type⟩⟩
```

A particular specialization of the previous case is the evaluation of a delayable mapping for delayed arguments, which should automatically cause—as announced and expected—the creation of a new delayed mapping representing the given expression. But there is another motivation leading to that implementation which shall be given here.

The definition of mappings frequently bases on the composition of existing mappings, which should therefore be provided generically as a delayed mapping type object referring to 5 mapping type objects. When evaluated, it applies the first mapping to the results of evaluating the other 4 mappings with the given arguments.

Given fun type arguments, the composition can be expressed by a fun specification as well, whose implementation does not have a state nor bears any surprises.

⟨`ComposeFunSpec_`⟩≡
```
    template <class Fun0_, class Fun1_, class Fun2_, class Fun3_, class Fun4_>
    struct ComposeFunSpec_
    {
        template <typename Param1_, typename Param2_,
                  typename Param3_, typename Param4_>
        struct Call
        {
        private:
            typedef Bind
            <   Exp<Fun1_>,Exp<Param1_>,Exp<Param2_>,Exp<Param3_>,Exp<Param4_>
            > Bind1_;
            typedef Bind
            <   Exp<Fun2_>,Exp<Param1_>,Exp<Param2_>,Exp<Param3_>,Exp<Param4_>
            > Bind2_;
            typedef Bind
            <   Exp<Fun3_>,Exp<Param1_>,Exp<Param2_>,Exp<Param3_>,Exp<Param4_>
            > Bind3_;
            typedef Bind
            <   Exp<Fun4_>,Exp<Param1_>,Exp<Param2_>,Exp<Param3_>,Exp<Param4_>
            > Bind4_;
            typedef Bind
            <   Exp<Fun0_>,Bind1_,Bind2_,Bind3_,Bind4_
            > Bind_;
        public:
            typedef typename Bind_::Result Result;
            static Result result(Param1_ param1, Param2_ param2,
                                 Param3_ param3, Param4_ param4)
            {
                return Bind_::result
                       (Fun0_(),
                        Bind1_::result(Fun1_(),param1,param2,param3,param4),
                        Bind2_::result(Fun2_(),param1,param2,param3,param4),
                        Bind3_::result(Fun3_(),param1,param2,param3,param4),
                        Bind4_::result(Fun4_(),param1,param2,param3,param4));
            }
        };
    };
```

The tricky part, though, is the invocation of this construct. It requires 5 arguments and therefore excludes an explicit instantiation by the evaluation of a named fun called, e.g., `Compose`. Instead, the creation of a composition has to be directly initiated by `Dispatch` which itself performs like an immediate mapping, as its evaluation respects, but is not guided by arguments being delayed or not. In fact, this quite harmless specialization finally realizes automatic delaying of a delayable mapping by creating the composition of the participating mappings as a delayed mapping.

⟨*delayable fun with delayed meta arguments*⟩≡

```
    template <Binding BINDING0_, class Specification0_,
              class Specification1_, class Specification2_,
              class Specification3_, class Specification4_>
    struct Dispatch<INTERNAL_META,true,true,
                    Internal<META,BINDING0_,Specification0_>,
                    Internal<META,DELAYED,Specification1_>,
                 Internal<META,DELAYED,Specification2_>,
                 Internal<META,DELAYED,Specification3_>,
                 Internal<META,DELAYED,Specification4_> >
  : internal_::MapSpecExp<Make<Exp<Internal
    <META,
     DELAYED,
     internal_::ComposeFunSpec_<Internal<META,BINDING0_,Specification0_>,
                                Internal<META,DELAYED,Specification1_>,
                                Internal<META,DELAYED,Specification2_>,
                                Internal<META,DELAYED,Specification3_>,
                                Internal<META,DELAYED,Specification4_> >
    > > > >
  {};
```

Again, if the arguments are (maps) of soma type, i.e., if they have a state, they have to be stored to have this state available when the member function `result` is actually called.

***Remark 9.5***
*Within the composition, the evaluations of the nested funs are virtually performed in parallel, as no definite order of evaluating the arguments of a function call is guaranteed. While this effect is unfavourable in the presence of side effects, an obvious extension of this implementation would be to truly perform these evaluations in parallel in a multiprocessor environment.*

*If the generic composition is consequently applied in mapping definitions, this immediately causes that all independent mapping evaluations are executed in parallel, without further adaption. While the realization of this idea would naturally require several dependent problems to be solved, the general advantage is to have a single point where to apply such technical modifications.*

*A less dramatic, but as well very helpful application to be implemented for composition instances and just for them would be some logging device, which informs on mapping evaluations or reports thrown exceptions and which would be implicitly apply to all mapping evaluations if mappings are generally composed by automatic delaying.*

## 9.4 Non-parallel evaluation

⟨*non-parallel evaluation*⟩≡
  ⟨`Sequential`⟩
  ⟨*finalizing funs*⟩
  ⟨`Cond`⟩
  ⟨⟨`CondAnd`⟩⟩
  ⟨⟨`CondOr`⟩⟩

The previous remarks motivate quite directly the next deliberation. Some mappings' arguments should not be evaluated in parallel but in a certain order or under certain conditions, not influencing the mapping's result but the choice and order of side effects. While arguments to according operators, like operator `? :` or operator `&&` are evaluated in an appropriate order on language level, this behaviour can be achieved by user-defined mappings only if applying them to delayed mapping arguments. Actually, such mappings could lead to dangerous misunderstandings if they were applicable to actual arguments and if they evaluated all of them in arbitrary order, while providing the familiar operator appearance. Thus, such mappings are defined to be of

- `DELAYED` domain, not accepting actual arguments, and
- `IMMEDIATE` effect, so that their definition is not subject to automatic delaying by composition, implying parallel evaluation.

Usually, one may define alternative mappings that have the same semantics concerning the result produced, but that evaluate their arguments in parallel. As an example, operator `&&` has the counterpart of a mapping that returns the result of evaluation a boolean AND on its arguments as well, but enforces both arguments to be (virtually) simultaneously evaluated.

### 9.4.1 Sequential evaluation

```
⟨Sequential⟩≡
    namespace internal_
    {
        ⟨SequentialFunSpec_⟩
    }
    ⟨⟨Sequential with meta type arguments⟩⟩
    ⟨⟨Sequential with any soma type argument⟩⟩
```

An exception to the last paragraph of the introduction—which is a mapping whose only purpose is to determine the order of side effects and which could therefore not be replaced by a variant operating on actual arguments—is the sequential evaluation operator `,` and the mapping `Sequential` represented by it.

```
⟨named fun classifications⟩+≡
    Named__(IMMEDIATE,2,Sequential,SOMA,sequential,DELAYED,
            Infix,COMMA,,DELAYED,Rvalue)
```

This implies that the builtin operator `,` applies to all non-delayed arguments and provides the appropriate behaviour, which is achieved by enforcing the operator overload to be defined only for `DELAYED` arguments. The result is a delayed mapping which evaluates the delayed mappings given as arguments in the desired order, implemented by the fun specification `SequentialFunSpec_`.

```
⟨SequentialFunSpec_⟩≡
    template <class Fun1_, class Fun2_>
    struct SequentialFunSpec_
    {
        template <typename Param1_, typename Param2_,
                  typename Param3_, typename Param4_>
        struct Call
        {
        private:
            typedef Bind<Exp<Fun1_>,
                         Exp<Param1_>,Exp<Param2_>,Exp<Param3_>,Exp<Param4_> >
                    Bind1_;
            typedef Bind<Exp<Fun2_>,
                         Exp<Param1_>,Exp<Param2_>,Exp<Param3_>,Exp<Param4_> >
                    Bind2_;
        public:
            typedef typename Bind2_::Result Result;
            static Result result(Param1_ param1, Param2_ param2,
                                Param3_ param3, Param4_ param4)
            {
                Bind1_::result(Fun1_(),param1,param2,param3,param4);
                return Bind2_::result(Fun2_(),param1,param2,param3,param4);
            }
        };
    };
```

The specialization of `Dispatch` is analogous to those above.

### 9.4.2 Conditional evaluation

⟨`Cond`⟩≡
```
    namespace internal_
    {
        ⟨CondFunCallDispatch_⟩
        ⟨CondFunSpec_⟩
    }
    ⟨⟨Cond with meta type arguments⟩⟩
    ⟨⟨Cond with any soma type argument⟩⟩
```

The mapping `Cond` corresponds to operator `? :` , which is the expression equivalent to an `if` statement. Its three arguments are required to be of delayed mapping type, as is the result.

⟨*named fun classifications*⟩+≡
```
    Named__(IMMEDIATE,3,Cond,SOMA,cond,DELAYED,,,,,)
```

The resulting mapping will be evaluated as follows: If the evaluation of the first mapping with the given arguments produces some result that represents a boolean TRUE, the result is given by the evaluation of (only) the second, otherwise of (only) the third mapping. The implementation of the according fun specification therefore depends on the stage of the first mapping's semantics. Given meta semantics, the condition is evaluated statically, and dynamically for soma semantics.

***Problem 9.6***
*The implementation of* `Cond` *should be independent of the particular representation of the condition's result and only respect the represented boolean value and its stage. This knowledge can only be provided by the representation itself, though, wherefore some means of decoupling is required.*

***Solution 9.7 (to problem 9.6)***
*In order to implement mappings whose behaviour invariantly bases on such fundamental information as rvalues of arithmetic type, the principle of all mappings having a generic return type has to be relaxed by allowing* finalizing mappings *to have a determined result type which just depends on the stage of the argument. The semantics of such a mapping is that of a cast of the argument to the according fundamental representation.*

Actually, there are three finalizing mappings: `ToBoolean`, `ToNatural` and `ToInteger`, which are introduced as individual mappings, as their implementation depends on the argument type implementation.

⟨*named fun classifications*⟩+≡
```
    Named__(INDIVIDUAL,1,ToBoolean,SOMA,toBoolean,ANY,,,,,)
    Named__(INDIVIDUAL,1,ToNatural,SOMA,toNatural,ANY,,,,,)
    Named__(INDIVIDUAL,1,ToInteger,SOMA,toInteger,ANY,,,,,)
```

Their result will be an instance of `Integral` if the argument has meta semantics, or of the according of `boolean`, `natural` or `integer` if it has soma semantics. For fundamental character and floating point types, such mappings have not been needed and provided yet.

⟨*finalizing funs*⟩≡

```
    #define Finalize__(Uppercase__,lowercase__)                                 \
    template <Binding BINDING0_, lowercase__ VALUE1_>                            \
    struct Dispatch                                                              \
    <INTERNAL_META,true,false,                                                   \
     Internal<META,BINDING0_,Named<INDIVIDUAL,1ul,To##Uppercase__##Tag> >,       \
     Internal<META,ACTUAL,Integral<lowercase__,VALUE1_> >                        \
     Fill1__(Void)                                                               \
    >                                                                            \
    : internal_::MapSpecExp                                                      \
      <Make<Exp<Internal<META,ACTUAL,Integral<lowercase__,VALUE1_> > > >         \
      >                                                                          \
    {};

    Finalize__(Boolean,boolean)
    Finalize__(Natural,natural)
    Finalize__(Integer,integer)

    #undef Finalize__


    #define Finalize__(Uppercase__,lowercase__,type__)                          \
    template <Binding BINDING0_>                                                 \
    struct Dispatch                                                              \
    <BUILTIN,true,false,                                                         \
     Internal<META,BINDING0_,Named<INDIVIDUAL,1ul,To##Uppercase__##Tag> >,       \
     type__                                                                      \
     Fill1__(Void)                                                               \
    >                                                                            \
    {                                                                            \
        struct Result                                                            \
        {                                                                        \
            template <typename Param0_, typename Param1_ Fill1__(typename) >     \
            struct Apply                                                         \
            {                                                                    \
                typedef lowercase__ Result;                                      \
                static Result result(Param0_, Param1_ param1 Fill1__(Void))      \
                {                                                                \
                    return param1;                                               \
                }                                                                \
            };                                                                   \
        };                                                                       \
    };

    Finalize__(Boolean,boolean,bool)
    Finalize__(Natural,natural,unsigned short)
    Finalize__(Natural,natural,unsigned int)
    Finalize__(Natural,natural,unsigned long)
    Finalize__(Integer,integer,signed short)
    Finalize__(Integer,integer,signed int)
    Finalize__(Integer,integer,signed long)

    #undef Finalize__
```

As `ToBoolean` has been defined by now, due to the common definition of named mappings in § 8.2, the fun specification for `Cond` dispatches its call according to the stage of the condition's result.

⟨`CondFunCallDispatch_`⟩≡

```
    template <typename Result_, class Fun1_, class Fun2_, class Fun3_>
    struct CondFunCallDispatch_;
```

⟨`CondFunSpec_`⟩≡

```
    template <class Fun1_, class Fun2_, class Fun3_>
    struct CondFunSpec_
    {
        template <typename Param1_, typename Param2_,
                  typename Param3_, typename Param4_>
        struct Call
        : CondFunCallDispatch_
          <typename ToBoolean<Exp
           <typename Bind<Exp<Fun1_>,Exp<Param1_>,Exp<Param2_>,
                                 Exp<Param3_>,Exp<Param4_> >::Result
           > >::Result,
           Fun1_,Fun2_,Fun3_
          >::template Call<Param1_,Param2_,Param3_,Param4_>
        {};
    };
```

If it has meta semantics and therefore is either of `True` or `False`, the call is forwarded directly to the according of the other two delayed mapping arguments. This means that the application of `Cond` at the meta stage encapsulates the technique of conditional type selection motivated in § 4.1.

⟨`CondFunCallDispatch_`⟩+≡

```
    template <class Fun1_, class Fun2_, class Fun3_>
    struct CondFunCallDispatch_<True,Fun1_,Fun2_,Fun3_>
    : Fun2_
    {};
    template <class Fun1_, class Fun2_, class Fun3_>
    struct CondFunCallDispatch_<False,Fun1_,Fun2_,Fun3_>
    : Fun3_
    {};
```

Otherwise, the result is given as a `bool` value at the soma stage, and the implementation encapsulates the conditional operator `? :`. This case therefore requires to explicitly answer the very tricky question of which type to return given that the two mappings in question do not evaluate to the same type, as stated for operator `? :` in [ISO98] § 5.16. This decision is performed by the mapping `Common` to be defined in § 11.1.2. As that mapping will have meta semantics, there is no danger of entering a circular dependency by basing its implementation again on this fun specification.

⟨`CondFunCallDispatch_`⟩+≡
```
    template <class Fun1_, class Fun2_, class Fun3_>
    struct CondFunCallDispatch_<bool,Fun1_,Fun2_,Fun3_>
    {
        template <typename Param1_, typename Param2_,
                  typename Param3_, typename Param4_>
        struct Call
        {
        private:
            typedef Bind
            <   Exp<Fun1_>,Exp<Param1_>,Exp<Param2_>,Exp<Param3_>,Exp<Param4_>
            > Bind1_;
            typedef Bind
            <   Exp<Fun2_>,Exp<Param1_>,Exp<Param2_>,Exp<Param3_>,Exp<Param4_>
            > Bind2_;
            typedef Bind
            <   Exp<Fun3_>,Exp<Param1_>,Exp<Param2_>,Exp<Param3_>,Exp<Param4_>
            > Bind3_;
        public:
            typedef typename Common<Bind2_,Bind3_>::Result Result;
            static Result result(Param1_ param1, Param2_ param2,
                                 Param3_ param3, Param4_ param4)
            {
                return
                    toBoolean(Bind1_::result(Fun1_(),param1,param2,param3,param4)) ?
                    Bind2_::result(Fun2_(),param1,param2,param3,param4) :
                    Bind3_::result(Fun3_(),param1,param2,param3,param4);
            }
        };
    };
```
The `Dispatch` specialization for `Cond` is analogous to that for `Sequential` and therefore not outlined. Equivalently to the mapping `Cond` realizing the functionality of operator `? :` , the mappings `CondAnd` and `CondOr` may be implemented, which provide an analogon to the operators `&&` and `||` with particular respect of their conditional argument evaluation.

⟨*named fun classifications*⟩+≡
```
    Named__(IMMEDIATE,2,CondAnd,SOMA,condAnd,DELAYED,
            Infix,DIRECT,&&,DELAYED,Rvalue)
    Named__(IMMEDIATE,2,CondOr,SOMA,condOr,DELAYED,
            Infix,DIRECT,||,DELAYED,Rvalue)
```
Given two delayed mappings ***dm1*** and ***dm2***, they may be expressed symbolically as

    cond(*dm1*,*dm2*,FALSE)

and

    cond(*dm1*,TRUE,*dm2*)

where `TRUE` and `FALSE` have to be interpreted as delayed mappings returning the according meta or soma object. Instead of exploiting this representation by introducing an appropriate conversion of meta to soma boolean values, a straightforward implementation according to that of `Cond` is chosen, which therefore does not have to be shown here. There are further mappings performing logical operations, which depend on the particular representation of the arguments, though, and always evaluate both arguments in arbitrary order. The prefix `Bool...` is used for identifiers of boolean mappings, which return the same result like their conditional counterparts but differ in the aspect of argument evaluation, and `Bit...` indicates bitwise operations on arguments representing a sequence of bits.

| **operation** | **conditional** `Cond...` | **boolean** `Bool...` | **bitwise** `Bit...` |
|---|---|---|---|
| `Not` | n.a. | `!` | `~` |
| `And` | `&&` | *named* | `&` |
| `Or` | `\|\|` | *named* | `\|` |
| `Xor` | n.a. | *named* | `^` |

Table 9.1: Logical operations

In table 9.1, the different mappings and the operators representing them are set in relation. While "n.a." means that there is no according mapping, *named* indicates that there is an identifier but no operator representing that mapping.

***Remark 9.8***
*A mapping* `CondNot` *would just behave like* `BoolNot`*, while the definition of a* `CondXor` *would not make sense, as XOR always has to evaluate both arguments to obtain its result.*

***Remark 9.9***
*The prefices are necessary to avoid conflicts with the builtin aliases* `not/and/or` *for boolean operators and* `compl/xor` *for bitwise operators, [ISO98] § 2.5.*

# Chapter 10

# Semantics

⟨*implementation*⟩+≡
    ⟨*general mappings*⟩
    ⟨*individual funs*⟩

The next property of a mapping to consider when determining how to evaluate it is the way its semantics depends on the argument types. This is meant in the sense of whether there is just a single implementation of this mapping to cover any argument type tuple, or whether it is feasible to provide different versions to respect particular types or type sets. This does not refer to the type category, though, as this provides a formal classification of types which surely has to be considered to decide how to evaluate mappings bound with them, but which does not describe of the types' semantics. This differenciation comes into play once that the consideration of a mapping to be evaluated at the meta or soma stage or to delay it has been accomplished and the mere mapping semantics determines the further steps to be taken.

## 10.1 General semantics

⟨*general mappings*⟩≡
    ⟨`Create`⟩
    ⟨`Param`⟩
    ⟨`Pick`⟩

A mapping with *general semantics* shall be called *general mapping* for brevity, which means that it refers to its parameters without consideration of the particular types' semantics and without regarding the type being internal, external or builtin. Such a mapping may be provided by a single definition or by a fixed set of them, but the relevant point is to state that no substantially different alternative could be feasible at all.

Of course, this has favourable consequences for its implementation, which can be accomplished by a single fun specification instanciated by default evaluation or by a specialization of `Dispatch` to account for the category and other externally available properties. A mapping by default is considered to be general, the reason of which will be explained in § 10.2.

### 10.1.1 `Create`

The first example again deals with the problem of the creation of an object, but unlike `Make` it is not meant to be evaluated if bound with a delayed argument type, but to be delayed itself.

⟨*named fun classifications*⟩+≡
```
    Named__(GENERAL,1,Create,META,create,ANY,,,,,)
```
Therefore, it is not an immediate mapping, but it is implemented easily in terms of the previous, taking advantage of the default evaluation accessing directly the mapping's fun specification.

⟨`Create`⟩≡
```
    template <>
    struct Named<GENERAL,1ul,CreateTag>
    {
        template <typename Param1_ Fill1__(typename)>
        struct Call
        : Make<Exp<Param1_> >
        {};
    };
```

### 10.1.2 `Param`

⟨`Param`⟩≡
```
    namespace internal_
    {
        ⟨ParamFunSpec_⟩
        ⟨ParamFunSpec_<N<N> >⟩
        ParamFunSpec__(1)
        ParamFunSpec__(2)
        ParamFunSpec__(3)
        ParamFunSpec__(4)
        #undef ParamFunSpec__
    }
    ⟨⟨Dispatch for Param with meta type argument⟩⟩
    ⟨P<N> and p(N)⟩
```
Besides creational mappings, the most frequently employed source of delayed funs is the mapping `Param`, which allows to specify a parameter to be unbound. This fun takes an argument representing a positive integer value $N$ and returns a delayed fun, which again returns the $N$th of its (arbitrary many, but at least $N$) arguments.

⟨*named fun classifications*⟩+≡
```
    Named__(GENERAL,1,Param,META,param,ANY,,,,,)
```
Being delayed, the result of `Param` serves as a placeholder for a parameter that is to be submitted when finally evaluating the delayed mapping. Therefore, the result serves as leaf of delayed expression trees.

Similar to the cases of conditional evaluation, this mapping considers the value represented by its argument, but not the representation itself. It seems awkward to consider `Param` a general mapping, but it is, as it leaves it to the (non-general) finalizing mapping `ToNatural` to provide the according rvalue of prescribed type. Again, the fun specification has to be parameterized by the type of the result to detect its state and, if that is meta, its value.

⟨`ParamFunSpec_`⟩≡
```
    template <typename ToNatural_>
    struct ParamFunSpec_;
```

For all feasible meta stage values, specializations are provided by a macro, each just returning the *N*th argument.

⟨ParamFunSpec_<N<*N*> >⟩≡
```
    #define ParamFunSpec__(N__)                                                    \
    template <>                                                                    \
    struct ParamFunSpec_<N<N__>::Result>                                           \
    {                                                                              \
        template <typename Param1_, typename Param2_,                              \
                  typename Param3_, typename Param4_>                              \
        struct Call                                                                \
        {                                                                          \
            typedef Param##N__##_ Result;                                          \
            static Result result(Param1_ param1, Param2_ param2,                   \
                                 Param3_ param3, Param4_ param4)                   \
            {                                                                      \
                return param##N__;                                                 \
            }                                                                      \
        };                                                                         \
    };
```

Like in previous cases, one might also want to provide a soma return value variant which allows the run-time specification of the argument to param, applying a `switch`, for instance, and using `Common` to convert any argument to a common return type. This is a rather hairy task, though, while its application did not appear to be of real interest, which is why the implementation has not been provided so far.

Again, the according `Dispatch` specialization does not have to be outlined. To abbreviate the notation of binding this mapping with a meta expression representing a meta natural value, macros are introduced which provide the notation `P<`*N*`>` and `p(`*N*`)` to express the placeholder $\mathtt{p}_N$ introduced in § 5.2.5 at the meta and the soma stage.

⟨P<*N*> *and* p(*N*)⟩≡
```
    template <natural N_>
    struct P
    : Param<N<N_> >
    {};
    #define p(N_) P<N_>::Result()
```

### 10.1.3 Pick

Similar to `Create`, there is a general alternative `Pick` to `Cond`, which performs a plain selection analogously to `If`. `Pick` does not require its arguments to be delayed; if they are, automatic delaying is performed. Accordingly, all arguments are evaluated.

⟨*named fun classifications*⟩+≡
```
    Named__(GENERAL,3,Pick,SOMA,pick,ANY,,,,,)
```

Still, the mapping is not provided explicitly for each (first) argument type representing a boolean value, but relies on `ToBoolean` to obtain an fundamental boolean rvalue. This is because the implementation has to be carried out either by template specializations or by the operator `? :` which cannot be abstracted.

Given that, the implementation is just a simplification of that of `Cond` and can be easily provided by means of instances of `Param`.

⟨Pick⟩≡
```
    template <>
    struct Named<GENERAL,3,PickTag>
    : Cond<P<1>,P<2>,P<3> >
    {};
```

## 10.2 Individual semantics

⟨*individual funs*⟩≡
  ⟨*fun stack reduction*⟩
  ⟨*individual fun call*⟩

In contrast to a general mapping, an *individual mapping* may be implemented in different ways, eventually considering individual properties of particular argument types. This can be turned into a definition: A mapping is individual if it is not general, i.e., if it can not be defined in a way that is guaranteed to hold and to be optimal for any (user-defined) type. This includes all mappings which could depend on the state of a soma argument object, but it excludes meta mappings which give information on the argument being of some type category (which may be obtained in a non-intrusive way) or of some builtin type (as their set is fixed and can be covered once and for all). Consequently, an individual mapping can be implemented in terms of a general one, but not vice versa.

According to the system of mapping representations introduced in the previous and to the declaration of general semantics being the default case, an individual mapping necessarily is a named mapping, while the converse does not hold. Therefore, mappings were considered to have general semantics by default. This is according to the necessity of naming to identify implementations of an individual mapping in the context of some type's definition, e.g., as a member.

### 10.2.1 Implementation choice

The attempt to evaluate an individual mapping may lead to a set of alternative implementations to consider, of which several may provide an `Undefined` result. If all do, the mapping evaluation has an `Undefined` result. Otherwise, the first implementation of defined result is chosen.

***Remark 10.1***
*One would expect that an unique solution were required and ambiguities should be considered illegal. Different implementations of an individual (non-unary) mapping may be provided, though, based on different sets of mappings whose evaluation is defined for the given arguments. If the arguments allow two or more alternatives to be applied, there is no reason to consider this to be illegal, but just an over-determination which should not bear problems, as long as all implementations provide the same mapping semantics. Otherwise, the individual implementation would be erroneous, not the fact of it being chosen.*

***Remark 10.2***
*Of course, alternative implementations may differ considerably by their performance. An essential improvement of the given ad-hoc selection process would be to select the most efficient*

*among the valid implementations. This requires compile-time information on the performance (in both run time and space) of a particular mapping evaluation, which is not provided in this approach so far but which might be added without corrupting the overall design.*

For the lack of an actual solution, the user has to take care herself of avoiding the choice of an inefficient implementation. Within the `STL` approach, the maximal complexity is considered part of the mapping's semantics. For instance, the random access operator `[]` for a container is required to be of constant cost, [ISO98] § 23.1.1;12.

This method could be imitated, but is not always sufficient, as the decision of an implementation to choose to evaluate a mapping may depend strongly on the fact whether the operation is of constant, linear or other complexity. As a consequence, different mappings would have to be defined, each guaranteeing a certain upper complexity bound (and implying higher bounds). The random access operator `[]` could be required to have constant costs, while the access to the $n$th element of a sequence at (maximally) linear costs could be provided by a different mapping called `ElementAt`. For the implementation of a sequence by a linked list only the latter mapping would be defined, an implementation by consecutively allocated memory would express the latter by the former.

***Problem 10.3***
*This method has the disadvantage of introducing a set of functionally equivalent versions of each mapping, each guaranteeing a particular complexity class. For instance, the access to the $n$th element of a sequence may also be obtained at logarithmic costs or even at quadratic costs in the case of an implicitly given sequence. Each complexity class would have to be reflected by an according mapping name, which would not only exhaust the implementor's phantasy, but also spoil expressiveness.*

***Solution 10.4 (to problem 10.3)***
*A senseful compromise is to provide the performance information separately, without influencing the description of a mapping's semantics. For the given example, the random access operator `[]` should just be described as returning a representation of the $n$th element of its first argument, where $n$ is the natural number represented by the second argument. Additionally, separate meta semantics mappings like `IsConstantRandomAccess` return the appropriate meta boolean value given those two arguments. The outcome of its evaluation can be used to qualify the implementation of another mapping to perform random access only if this prerequisite holds.*

This is not a very beautiful solution, but practicable to serve as intermediate solution. It relaxes the requirements given in remark 10.2, as it does not have to suit automatic evaluation and therefore does not have to be provided in a complete manner, but just for those cases where the choice of an implementation shall be explicitly restricted by performance criteria.

### 10.2.2 Fun stacks

⟨*fun stack reduction*⟩≡
```
namespace internal_
{
    ⟨InsertFrontDefined_⟩
    ⟨SelectDefined_⟩
    ⟨GetFront_⟩
    ⟨StackExpFunExp_⟩
}
```

Due to the previous, given a set of fun definitions of equal semantics and given a tuple of arguments, that subset of the former has to be selected at compile-time all elements of which evaluate to a defined result for those arguments. Of this, the first element is chosen.

A meta stack of fun type meta expressions serves as representation of such a set. The process is implemented in a direct way, not using the `Bind` / `Dispatch` mechanism, in order to avoid circular dependencies. Because this stack is not intended to be used as a soma object, just the meta stack specification `Stack<...>` is used for its representation instead of its complete definition as the internal meta type `Internal<META,ACTUAL,Stack<...> >`.

The main step of the process is performed by the template class `SelectDefined_`, which takes a given fun expression stack and parameter expressions as arguments and returns a new stack of those fun expressions whose call is defined for these parameters. If the given stack is empty, the result is as well. Otherwise, its first element is inserted at the front of the recursively obtained result, if its call for the arguments given by the parameter expressions is defined.

```
⟨SelectDefined_⟩≡
    template <class FunExpStack_,
              class Exp1_, class Exp2_, class Exp3_, class Exp4_>
    struct SelectDefined_;

    template <class Front_, class EraseFront_,
              class Exp1_, class Exp2_, class Exp3_, class Exp4_>
    struct SelectDefined_<Stack<Front_,EraseFront_>,Exp1_,Exp2_,Exp3_,Exp4_>
    : InsertFrontDefined_
      <typename Bind<Front_,Exp1_,Exp2_,Exp3_,Exp4_>::Result,
       Front_,
       typename SelectDefined_<EraseFront_,Exp1_,Exp2_,Exp3_,Exp4_>::Result
      >
    {};

    template <class Exp1_, class Exp2_, class Exp3_, class Exp4_>
    struct SelectDefined_<Empty,Exp1_,Exp2_,Exp3_,Exp4_>
    {
        typedef Empty Result;
    };
```

The step mentioned last is performed by the helper meta function `InsertFrontDefined_`, which is given by a simple dispatch.

```
⟨InsertFrontDefined_⟩≡
    template <typename Result_,
              class FunExp_, class DefinedFunExpStack_>
    struct InsertFrontDefined_
    {
        typedef Stack<FunExp_,DefinedFunExpStack_> Result;
    };

    template <class FunExp_, class DefinedFunExpStack_>
    struct InsertFrontDefined_<Undefined,FunExp_,DefinedFunExpStack_>
    {
        typedef DefinedFunExpStack_ Result;
    };
```

The obtained stack serves as argument to the meta function `GetFront_` which returns the fun given by the first element of the stack and `UndefinedFun` if that is empty.

```
⟨GetFront_⟩≡
    template <class FunExpStack_>
    struct GetFront_;

    template <typename Front_, class EraseFront_>
    struct GetFront_<Stack<Front_,EraseFront_> >
    : Front_
    {};

    template <>
    struct GetFront_<Empty>
    : Exp<UndefinedFun>
    {};
```

The entire process is wrapped within a fun expression `StackExpFunExp_` which is parameterized by a meta expression providing the stack. Effectively, the stack is filtered to a single choice, if existent. A better selection mechanism could be implemented by just replacing `GetFront_` by some other selection method.

```
⟨StackExpFunExp_⟩≡
    template <class FunExpStackExp_>
    struct StackExpFunExp_
    {
        struct Result
        {
            template <typename Param1_, typename Param2_,
                      typename Param3_, typename Param4_>
            struct Call
            : GetFront_<typename SelectDefined_
              <typename FunExpStackExp_::Result,
               Exp<Param1_>,Exp<Param2_>,Exp<Param3_>,Exp<Param4_>
              >::Result>::Result
              ::template Call<Param1_,Param2_,Param3_,Param4_>
            {};
        };
    };
```

### 10.2.3 Argument-specific implementation

```
⟨individual fun call⟩≡
    namespace internal_
    {
        ⟨MemberFunExp⟩
    }
    ⟨FunExpStackExp⟩
    ⟨Dispatch for individual fun call⟩
```

An individual mapping definition may be specialized for a particular internal soma type argument by an according member fun of that type. This should not be done unless necessary, though, as that parameter would not be generic in the definition anymore, but be bound to this precise type. Usually, it makes sense only if the definition involves direct access to member data of the argument. This should not be confused with the decision to generally evaluate mappings globally—the (type-dependent) definition may be provided as a member, but may not be called directly, because common treatment of mapping evaluation could not be provided then. On the other hand, the possibility of providing the definition as a member saves the object-oriented programming principle of formally associating (truly) type-dependent information with the type—only the call syntax has been globalized.

***Problem 10.5***
*Given a non-unary individual mapping, which argument is responsible for providing the fun definition member if the mapping is bound with different argument types?*

Basically, this is rather a question of design than of technique. By common practice, the "most complex" of the participating types will be responsible. E.g., the multiplication of a scalar and a vector should be defined on part of the vector—otherwise, a scalar would have to provide a myriad of definitions!

But the vector may the first or the second argument, wherefore the fun would have to be defined in two variants (of which one might just swap the arguments and call the other). Analogously to the discussion in § 7.1.1, the vector type should provide those as specializations

`Identifier<1>` and `Identifier<2>`

of a member class template parameterized by its position in the argument list.

To avoid the definition of numerous individual member templates, a single member template class `Individual` parameterized by the identifier (tag) and the position should be used, though, which by default is defined to have an `Undefined` result and whose specializations

`Individual<IdentifierTag,1>` and `Individual<IdentifierTag,2>`

provide the according implementations.

***Remark 10.6***
*The complete specialization of a member class template is not allowed if the enclosing (e.g., vector) class is given by a class template that is not completely specialized. This case is most probable, though, as demonstrated by the previous examples which were parameterized at least by the vector element type.*

*The solution is a rather cheap trick:* `Individual` *just takes a third dummy parameter which is set by default to any type, e.g.,* `Void`*. This way, the specializations are made partial, which does not conflict with the case of a partially or non-specialized enclosing class template.*

Like for the general consideration, one can and should not expect that there is just an unique implementation of that mapping for the given type being the type of the argument at the given position. Again, there may be a set of implementations depending on the other arguments, which would again be given by a meta stack.

***Solution 10.7 (to problem 10.5)***
*Any internal soma type contains a member*

```
template <class Tag_, natural INDEX_, typename Dummy_=Void>
struct FunExpStackExp;
```

*whose instances are meta stack expressions whose elements again are fun type expressions providing alternative implementations for the according combination of the indicated individual fun and an argument of that soma type at the indicated position of the argument list. The default definition of that template provides an empty stack.*

This means that the call ***`Identifier<Arg1,...,ArgN>`*** may be defined by either fun within the meta stacks

```
Arg1::FunExpStackExp<IdentifierTag,1>::Result
...
ArgN::FunExpStackExp<IdentifierTag,N>::Result
```

It is the responsibility of the application programmer to provide meaningful and corresponding definitions for these and to reduce redundancies, e.g., by keeping to the "most complex type" principle.

To translate the explicit member notation to a parameterized format, the construct `MemberFunExp` is used, which serves as an dispatch to detect whether the second argument has internal soma type and which returns the reduced form of the member fun stack according to the further parameters.

⟨MemberFunExp⟩≡

```
template <typename Stripped_, class Tag, natural INDEX_>
struct MemberFunExp
: Exp<UndefinedFun>
{};

template <class Specification_, class Tag_, natural INDEX_>
struct MemberFunExp<Internal<SOMA,ACTUAL,Specification_>,Tag_,INDEX_>
: StackExpFunExp_<typename Internal<SOMA,ACTUAL,Specification_>
                          ::template FunExpStackExp<Tag_,INDEX_> >
{};
```

Besides those member implementations, there will most likely be further implementations of the mapping which are independent of particular argument types, but which are defined globally in terms of other mappings whose evaluation must be defined for the given arguments in order to have a valid definition. For each individual mapping associated with a given `FunTag_`, a meta stack containing all of these general definitions as fun expressions will be provided by an instance of `FunExpStackExp` which by default returns an empty stack.

⟨FunExpStackExp⟩≡

```
template <class FunTag_>
struct FunExpStackExp
: Exp<Empty>
{};
```

Now, all the elements are put together to form the `Dispatch` instance responsible of finding a defined implementation of the evaluation of an individual mapping. Note that for each internal soma argument the according member fun stack is reduced to a single fun first, then these funs are put on top of the stack of general definitions. Due to the selection of the first defined implementation, the precedence of member implementations within argument types follows their order in the argument list and all of them are preferred to any general implementation.

⟨`Dispatch` *for individual fun call*⟩≡
```
    template<Level LEVEL_, Binding BINDING0_, natural ARITY0_, class Tag0_,
             typename Stripped1_, typename Stripped2_,
             typename Stripped3_, typename Stripped4_>
    struct Dispatch<LEVEL_,true,false,
                    Internal<META,BINDING0_,Named<INDIVIDUAL,ARITY0_,Tag0_> >,
                    Stripped1_,Stripped2_,Stripped3_,Stripped4_>
    : internal_::MapSpecExp<internal_::StackExpFunExp_<Exp
      <Stack<internal_::MemberFunExp<Stripped1_,Tag0_,1>,
       Stack<internal_::MemberFunExp<Stripped2_,Tag0_,2>,
       Stack<internal_::MemberFunExp<Stripped3_,Tag0_,3>,
       Stack<internal_::MemberFunExp<Stripped4_,Tag0_,4>,
             typename FunExpStackExp<Tag0_>::Result> > > >
      > > >
    {};
```
In absence of internal soma type arguments, only general definitions are considered. To provide a specialized implementation for these cases, one has to provide an appropriate specialization of `Dispatch`.

***Remark 10.8***
*Instead of referring to a stack of alternative fun definitions, the decision process could be encoded as a delayed fun itself. This would allow to perform run-time decisions as well, i.e., to dynamically select the implementation which is most efficient for data given at run-time.*

## 10.3 Individual semantics operators

⟨*implementation*⟩+≡
    ⟨`OperatorTraits`⟩
    ⟨*operator with builtin type arguments*⟩
    ⟨*individual mapping with non-internal and meta arguments*⟩
    ⟨*operator with meta arguments*⟩

For the subset of individual mappings which are represented by operators, the builtin implementation has to be provided if all arguments are of builtin type, but as well if they are of internal meta type, which just causes the operation to be performed at compile-time. While definitions applying the according operator to the represented constant value(s) can be provided in an automatized manner, the result types have to be specified explicitly, as that information is not provided by the language. This task is accomplished by the template `OperatorTraits`, which takes the according tag and one or two parameter types.

⟨`OperatorTraits`⟩≡
```
    template <class FunTag_, typename Param1_, typename Param2_=Void>
    struct OperatorTraits;
```
The instances of this template are expected to contain two nullary meta functions, `IsDefined` returning a meta boolean and `ToResult` returning a type. These informations have to be

given separately to avoid circular dependencies: Mappings on meta booleans are evaluated to find out whether operator applications are defined, but these operators again are used for the implementation of those mappings.

## 10.3.1 Builtin arguments

⟨*operator with builtin type arguments*⟩≡
```
    ⟨auxiliary types for operator with builtin type arguments⟩
    ⟨⟨prefix operator with builtin type argument⟩⟩
    ⟨⟨postfix operator with builtin type argument⟩⟩
    ⟨infix operator with builtin type arguments⟩
    ⟨⟨paired operator with builtin type arguments⟩⟩
    ⟨⟨macros for operator with builtin type arguments⟩⟩
    #include "named.hh"
    ⟨⟨undefine macros for operator with builtin type arguments⟩⟩
```
If an operator is applied to builtin type arguments, both informations provided by the `OperatorTraits` have to be merged to provide the result type, which is either `Undefined` if the member `IsDefined` says so, or the type provided by `ToResult` otherwise. This deduction is performed by an `OperatorTraitsDispatch` initiated by the auxiliary template class `Merge` in straightforward manner.

⟨*auxiliary types for operator with builtin type arguments*⟩≡
```
    namespace internal_
    {
        template <class IsDefined_, class OperatorTraits_>
        struct OperatorTraitsDispatch;
        template <class OperatorTraits_>
        struct OperatorTraitsDispatch<False,OperatorTraits_>
        : Exp<Undefined>
        {};
        template <class OperatorTraits_>
        struct OperatorTraitsDispatch<True,OperatorTraits_>
        : OperatorTraits_::template ToResult<>
        {};

        template <class OperatorTraits_>
        struct Merge
        : OperatorTraitsDispatch
          <typename OperatorTraits_::template IsDefined<>::Result,
           OperatorTraits_
          >
        {};
    }
```
Given the appropriate result type, the definition of an operator application is a mere issue of lexical replacement, which is displayed for the case of binary infix operators.

⟨*infix operator with builtin type arguments*⟩≡
```
    #define PositionInfix__(Id__,MODE__,OP__)                                   \
    template <Binding BINDING0_,                                                \
              typename Stripped1_, typename Stripped2_>                         \
    struct Dispatch<BUILTIN,true,false,                                         \
                    Internal<META,BINDING0_,Named<INDIVIDUAL,2ul,Id__##Tag> >,   \
                Stripped1_,Stripped2_ Fill2__(Void)>                            \
    {                                                                           \
        struct Result                                                           \
        {                                                                       \
            template <typename Param0_,                                         \
                      typename Param1_, typename Param2_ Fill2__(typename)>     \
            struct Apply                                                        \
            {                                                                   \
                typedef typename internal_::Merge                               \
                        <OperatorTraits<Id__##Tag,Stripped1_,Stripped2_>        \
                        >::Result                                               \
                        Result;                                                 \
                static Result result(Param0_ param0,                            \
                                     Param1_ param1, Param2_ param2 Fill2__(Void)) \
                {                                                               \
                    return param1 OP__ param2;                                  \
                }                                                               \
            };                                                                  \
        };                                                                      \
    };
```

These definitions are provided for all individual operators, dispatching their position, by macro specialization.

## 10.3.2 Non-internal and meta arguments

⟨*individual mapping with non-internal and meta arguments*⟩≡

```
    ⟨MetaToSoma⟩
    ⟨Dispatch for individual mapping with non-internal and meta arguments⟩
```

If an individual semantics mapping is bound with both non-internal and meta type arguments, the evaluation has to be performed at run-time. One might want to apply partial evaluation to, e.g., eliminate operations on a constant neutral element, but by default the constant should be replaced by its soma value. This is performed by the mapping `MetaToSoma`, which might as well be specialized to transform more complex meta objects to soma objects.

⟨*named fun classifications*⟩+≡

```
    Named__(GENERAL,1,MetaToSoma,SOMA,metaToSoma,ANY,,,,,)
```

This mapping is defined explicitly by a specialization of `Dispatch` in order to take advantage of the direct availability of the stripped argument type.

⟨`MetaToSoma`⟩≡

```
    ⟨⟨MetaToSoma and soma argument⟩⟩
    ⟨MetaToSoma and meta integral argument⟩
```

By default, the argument has soma semantics and `MetaToSoma` is implemented as the identity. For a meta integral argument, the builtin type value parameter is returned. Specializations for other meta data structures have to be added to transform them to a representation at the soma stage.

⟨`MetaToSoma` *and meta integral argument*⟩≡

```
    namespace internal_
    {
        template <typename type_, type_ VALUE_>
        struct MetaToSomaFunSpec_
        {
            template <typename Param1_ Fill1__(typename)>
            struct Call
            {
                typedef type_ Result;
                static Result result(Param1_ Fill1__(Void))
                {
                    return VALUE_;
                }
            };
        };
    }
    template <Binding BINDING0_, typename type1_, type1_ VALUE1_>
    struct Dispatch
    <INTERNAL_META,true,false,
     Internal<META,BINDING0_,Named<GENERAL,1ul,MetaToSomaTag> >,
     Internal<META,ACTUAL,Integral<type1_,VALUE1_> >
     Fill1__(Void)
    >
    : internal_::MapSpecExp<Exp<internal_::MetaToSomaFunSpec_<type1_,VALUE1_> > >
    {};
```

Now, the evaluation of an individual mapping with meta and soma arguments by default is implemented by transforming all arguments to their soma representation and returning the result of binding the mapping with these.

⟨`Dispatch` *for individual mapping with non-internal and meta arguments*⟩≡

```
    template <Binding BINDING0_, class Tag_,
              typename Stripped1_, typename Stripped2_,
              typename Stripped3_, typename Stripped4_>
    struct Dispatch<INTERNAL_META,true,false,
                    Internal<META,BINDING0_,Named<INDIVIDUAL,2ul,Tag_> >,
                    Stripped1_,Stripped2_,Stripped3_,Stripped4_>
    : internal_::MapSpecExp
      <Bind<Exp<Internal<META,BINDING0_,Named<INDIVIDUAL,2ul,Tag_> > >,
            MetaToSoma<P<1> >,MetaToSoma<P<2> >,
            MetaToSoma<P<3> >,MetaToSoma<P<4> > >
      >
    {};
```

## 10.3.3 Meta arguments

⟨*operator with meta arguments*⟩≡

    ⟨⟨*prefix operator specialization for actual meta parameter*⟩⟩
    ⟨⟨*postfix operator specialization for actual meta parameter*⟩⟩
    ⟨*infix operator specialization for actual meta parameters*⟩
    ⟨⟨*macros for operator specialization for actual meta parameters*⟩⟩
    `#include "named.hh"`
    ⟨⟨*undefine macros for operator specialization for actual meta parameters*⟩⟩

An operation on meta objects is the most elementary way to apply an operator and is integrated into the mapping evaluation machinery by a specialization of `Dispatch` for this particular type category. Of the meta types presented so far, operators apply directly only to Integral objects.

The application of the random access operator [] to a meta stack, for instance, would be defined instead by a global implementation stepping through the stack elements by means of some more elementary mapping and by counting the position, until the element to be returned is found. The same implementation could then be applied to a soma linked list object!

The specialization of an operator evaluation with Internal arguments, in contrast, can be composed of other mappings expressed directly by applying the operator to the constant value argument of Internal, which is illustrated here for the infix operator case.

⟨*infix operator specialization for actual meta parameters*⟩≡

```
    #define PositionInfix__(Id__,OP__)                                            \
    template <Binding BINDING0_,                                                  \
              typename type1_, type1_ VALUE1_,                                    \
              typename type2_, type2_ VALUE2_>                                    \
    struct Dispatch                                                               \
    <INTERNAL_META,true,false,                                                    \
     Internal<META,BINDING0_,Named<INDIVIDUAL,2ul,Id__##Tag> >,                   \
     Internal<META,ACTUAL,Integral<type1_,VALUE1_> >,                             \
     Internal<META,ACTUAL,Integral<type2_,VALUE2_> >                              \
     Fill2__(Void)                                                                \
    >                                                                             \
    : internal_::MapSpecExp<Make<Exp                                              \
      <Internal<META,ACTUAL,                                                      \
                Integral<typename OperatorTraits<Id__##Tag,type1_,type2_>         \
                         ::template ToResult<>::Result,                           \
                         (typename OperatorTraits<Id__##Tag,type1_,type2_>        \
                         ::template ToResult<>::Result) (VALUE1_ OP__ VALUE2_)> >  \
      > > >                                                                       \
    {};
```

The macro is called for all individual delayable operators applicable to rvalues, which excludes paired and modifying operators, see § 8.1.2, and those defined in § 9.4.1 and § 11.1.3.

# Chapter 11

# Named funs

Different named funs shall now be defined, both for illustration and to provide a base layer for further definitions. As a consequence of the previous, the identifiers and the calls of all named funs presented in the oncoming have already been defined and may therefore be used to define other funs, as long as circular dependencies are avoided. This holds particularly for funs whose definitions do not rely on any other construct at all.

## 11.1 General semantics

⟨*implementation*⟩+≡
  ⟨*trait type mappings*⟩
  ⟨*non-trait type mappings*⟩
  ⟨*general object mappings*⟩

### 11.1.1 Trait type mappings

⟨*trait type mappings*⟩≡
  ⟨⟨`StageMETA__`⟩⟩
  ⟨`StageSOMA__`⟩
  ⟨⟨*define macros for trait type mappings*⟩⟩
  ⟨⟨*named trait type fun classifications*⟩⟩
  ⟨⟨*undefine macros for trait type mappings*⟩⟩

Unary mappings on type arguments have been considered in § 6.2, where this class of mappings was implemented by collective traits, which may now be used to comfortably define these mappings as funs. Remind that only such properties were defined by traits which describe the argument's category or structural features originating from the C++ language specification. Other unary mappings are defined separately, as it would be tedious to extend the implementation of `Traits` for any of them.

The categorization of a type within the C++ type system is provided by predicates, i.e, soma stage mappings returning a meta boolean object, most of which are self-explanatory in combination with figures 6.1-6.3.

| | | |
|---|---|---|
| `IsReference` | `IsObjectReference` | `IsFunctionReference` |
| `IsPointable` | `IsVoid` | `IsReferenceable` |
| `IsFunction` | `IsObjectType` | `IsClass` |
| `IsArray` | `IsBoundArray` | `IsUnboundArray` |
| `IsScalar` | `IsPointer` | |
| `IsVoidPointer` | `IsObjectPointer` | `IsFunctionPointer` |
| `IsMemberPointer` | `IsMemberObjectPointer` | `IsMemberFunctionPointer` |
| `IsEnum` | `IsArithmetic` | `IsReal` |
| `IsFloat` | `IsDouble` | `IsLongDouble` |
| `IsIntegral` | `IsSigned` | `IsUnsigned` |
| `IsBool` | `IsChar` | `IsWchar` |
| `IsCharacter` | `IsSignedChar` | `IsUnsignedChar` |
| `IsShort` | `IsSignedShort` | `IsUnsignedShort` |
| `IsInt` | `IsSignedInt` | `IsUnsignedInt` |
| `IsLong` | `IsSignedLong` | `IsUnsignedLong` |
| `IsQualified` | `IsQualifiable` | `IsCallable` |
| `IsConst` | `IsVolatile` | `IsConstVolatile` |

Except for those mappings that detect qualification, eventual qualifiers are generally ignored. For example, `IsDouble` is true if its argument is any of

| | |
|---|---|
| `double` | `double const` |
| `double volatile` | `double const volatile` |

`IsClass` collectively detects `class`es, `struct`s, `union`s and bitfields, all of which are class constructs. There is no means to distinguish between them by C++ language elements. Hence, to ensure that any type identified as a class can be inherited, `union`s and bitfields have to be generally avoided or to be individually wrapped into appropriate classes.

Furthermore, there are pure meta stage mappings that map the argument type to associated types:

- `ToConst` returns the `const`-qualified argument type if that is qualifiable, or a reference to the `const`-qualified referenced type if the argument is a reference to a qualifiable type, otherwise (i.e., the argument is a function or function reference type) its instantiation is illegal.

- `ToVolatile` and `ToConstVolatile` analogously qualify their argument by `volatile` and `const volatile`.

- `ToNonConst` returns the argument type, an eventual `const`-qualifier being stripped off, which for references relates to the referenced type.

- `ToNonVolatile` and `ToNonQualified` analogously strip `volatile` or all qualifiers.

- `ToReference` returns the reference type to the argument type if it is referencable, or the argument type itself if it is a reference, otherwise (i.e., the argument is—possibly qualified—`void`) its instantiation is illegal.

- `ToNonReference` returns the type referenced by the argument if it is a reference, otherwise the argument type itself.

- `ToStripped` is the result of applying both `ToNonQualified` and `ToNonReference`.
- `ToPointer` returns the pointer type to the argument type if it is pointable, otherwise (i.e., the argument is a reference type) its instantiation is illegal.
- `ToNonPointer` returns the type pointed at by the argument if it is a pointer, otherwise the argument type itself; this mapping is not called `Pointee` deliberatively, as the result is not a reference.
- `ToPromoted` returns the (possibly wider) type to which an rvalue of the (arithmetic) argument type gets *promoted* when used in an arithmetic operation, [ISO98] § 4.5-6. It is illegal to instantiate this mapping by a non-arithmetic argument type.
- `ToPriority` returns for an arithmetic argument a meta natural number from `0` to `6` reflecting its level in the arithmetic conversion hierarchy according to [ISO98] § 5;9. This trait is only used for the purpose of performing such conversions, see § 11.1.2.
- `ToReturn` returns the result type if the argument type is callable, otherwise its instantiation is illegal.
- `ToParamStack` returns a meta stack holding the parameter types ordered from top to bottom if the argument type is callable, otherwise its instantiation is illegal.
- `ToClass` returns the class type to which the member belongs if the argument is a pointer to member, otherwise its instantiation is illegal.
- `ToMember` returns the member type if the argument is a pointer to member, otherwise its instantiation is illegal.
- `ToElement` returns the element type if the argument is an array type, otherwise its instantiation is illegal.
- `ToLength` returns the number of elements as a meta natural value if the argument is a bound array, otherwise its instantiation is illegal.

***Remark 11.1***
*Intentionally, there are no mappings* `IsMeta`*,* `IsSoma`*,* `IsDelayed` *and* `IsNonDelayed`*, as these properties are covered by the* `Bind` */* `Dispatch` *mechanism and shall not be considered within application code. For the same reason, a type's* `Binding`*,* `Level` *and* `Effect` *traits are not provided by mappings.*

Unary named type mappings are simply defined by directly accessing the information in the `Traits` class template. While the implementation of meta mappings is given by the mere provision of the according type member of the `Traits` instance, soma mappings involve the creation of the resulting meta object as a soma object.

⟨StageSOMA__⟩≡
```
    #define StageSOMA__(Id__)                                              \
    template <>                                                            \
    struct Named<GENERAL,1ul,Id__##Tag>                                    \
    {                                                                      \
        template <typename Param1_ Fill1__(typename)>                      \
        struct Call                                                       \
        : Bind<Make<Exp<typename internal_::Traits<Param1_>::Id__> >,    \
               Exp<Param1_> >                                             \
        {};                                                               \
    };
```

### 11.1.2 Non-trait type mappings

⟨*non-trait type mappings*⟩≡

```
    ⟨ToArray⟩
    ⟨IsSame⟩
    ⟨IsStatic⟩
    ⟨IsPolymorphic⟩
    ⟨IsConvertible⟩
    ⟨Common⟩
```

In contrast to the collective definition of trait mappings by a single class template, in the general case it is not possible to implement type mappings in this manner, for the usual application domain lacking such strict hierarchical structure.

Type mappings of general semantics are usually defined directly, using template specializations of `Dispatch` and eventual auxiliary dispatches.

#### `IsSame`

A very simple example of using this approach is the following type mapping determining whether two types are identical.

⟨*named fun classifications*⟩+≡

```
    Named__(GENERAL,2,IsSame,SOMA,isSame,ANY,,,,,)
```

The implementation employs a well-known auxiliary dispatch. It seems quite clumsy, but contains no particular difficulties.

⟨`IsSame`⟩≡

```
    namespace internal_
    {
        template <typename Type1_, typename Type2_>
        struct IsSameDispatch
        : B<false>
        {};
        template <typename Type_>
        struct IsSameDispatch<Type_,Type_>
        : B<true>
        {};
    }
    template <>
    struct Named<GENERAL,2ul,IsSameTag>
    {
        template <typename Param1_, typename Param2_ Fill2__(typename=Void)>
        struct Call
        : Bind<Make<internal_::IsSameDispatch<Param1_,Param2_> >,
               Exp<Param1_>,Exp<Param2_> >
        {};
    };
```

## ToArray

A more peculiar implementation is needed for the mapping `ToArray`, which, if given an object type as first and a meta natural as second argument, returns the bound array type of the specified number of elements of the first type.

⟨*named fun classifications*⟩+≡
```
    Named__(GENERAL,2,ToArray,META,toArray,ANY,,,,,)
```
Here, it is necessary to decide whether the mapping is defined, which is the case if the first argument is an object type. This can be expressed in terms of funs, thanks to the fun `Cond`.

⟨`ToArray`⟩≡
```
    namespace internal_
    {
        template <natural LENGTH_>
        struct ToArrayFun_
        {
            template <typename Param1_, typename Param2_ Fill2__(typename=Void)>
            struct Call
            : Exp<Param1_[LENGTH_]>
            {};
        };
    }
    template <Level LEVEL_, Binding BINDING0_,
              typename Stripped1_,
              natural LENGTH_>
    struct Dispatch<LEVEL_,true,false,
                    Internal<META,BINDING0_,Named<GENERAL,2ul,ToArrayTag> >,
                    Stripped1_,
                    Internal<META,ACTUAL,Integral<natural,LENGTH_> >
                    Fill2__(Void)>
    : internal_::MapSpecExp
      <Cond<IsObjectType<P<1> >,
            Bind<Exp<internal_::ToArrayFun_<LENGTH_> >,P<1>,P<2> >,
            Bind<Exp<UndefinedFun>,P<1>,P<2> > > >
    {};
```

## IsStatic

There are some rather elementary mappings that are defined in an intrusive manner, requiring argument types to be complete so that class members can be accessed or objects of that type may be declared. Both will be necessary to implement the following mappings. Their definitions are inspired by constructs which were presented in [Ale01].

`IsStatic` returns a meta boolean indicating whether the argument is an empty class.

⟨*named fun classifications*⟩+≡
```
    Named__(GENERAL,1,IsStatic,SOMA,isStatic,ANY,,,,,)
```
This does not only mean the absence of non-static member data, but usually of any virtual member function as well, as these are commonly implemented by means of an internal pointer within each object, therefore causing memory overhead.

To check whether a class is empty, the sizes of two classes are compared, which have some identical member data, but one of which inherits the given class. Since only empty base class subobjects are of zero size, [ISO98] § 1.8;5, conversely the class must be empty if both test classes have equal size.

⟨IsStatic⟩≡
```
    namespace internal_
    {
        struct Plain_
        {
            void* p;
        };
        template <class Class_>
        struct Derived_
        : Class_
        {
            void* p;
        };
        struct IsStaticFun_
        {
            template <typename Param1_ Fill1__(typename=Void)>
            struct Call
            : Bind<Make<B<(sizeof(Plain_) == sizeof(Derived_<Param1_>))> >,
                   Exp<Param1_> >
            {};
        };
    }
    template <>
    struct Named<GENERAL,1ul,IsStaticTag>
    : Cond<IsClass<P<1> >,
           Exp<Internal<META,DELAYED,internal_::IsStaticFun_> >,
           Make<B<false> > >
    {};
```
As the stripped parameter types do not have to be analyzed in this implementation, it is not necessary to provide a specialization of `Dispatch`, but just a definition of the fun specification.

## IsPolymorphic

`IsPolymorphic` returns a meta boolean value indicating whether the argument is a (dynamically) polymorphic class, i.e., whether it has virtual member functions, [ISO98] § 9;6, 9.2;2.

⟨*named fun classifications*⟩+≡
```
    Named__(GENERAL,1,IsPolymorphic,SOMA,isPolymorphic,ANY,,,,,)
```
Usually, virtual functions are realized by an internal pointer to the virtual function table that is stored within each object, accordingly augmenting its size. The argument class is therefore inherited by two non-empty classes, one of which declares a virtual function. If this does not lead to a difference in the size of objects of those two classes, the argument class itself already contains a virtual function table pointer and therefore virtual functions.

⟨IsPolymorphic⟩≡
```
    namespace internal_
    {
        template <class Class_>
        struct Virtual_
        : Class_
        {
            void* p;
            virtual ~Virtual_()
            {}
            virtual void f()
            {}
        };
        struct IsPolymorphic_
        {
            template <typename Param1_ Fill1__(typename=Void)>
            struct Call
            : Bind<Make<B<(   sizeof(Derived_<Param1_>)
                           == sizeof(Virtual_<Param1_>))> >,Exp<Param1_> >
            {};
        };
    }
    template <>
    struct Named<GENERAL,1ul,IsPolymorphicTag>
    : Cond<IsClass<P<1> >,
           Exp<Internal<META,DELAYED,internal_::IsPolymorphic_> >,
           Make<B<false> > >
    {};
```

***Remark 11.2***
*This test is not truly reliable, as the use of internal pointers to a virtual function table within objects is the common solution, but not dictated by the standard. As an explicit alternative, one might introduce a meta boolean value member* `IsPolymorphic` *in each class which would have to be set to* `True` *if virtual functions are defined or a polymorphic class is inherited.*

`IsConvertible`

The binary mapping `IsConvertible` does not need a specialization to be implemented; it rather uses previous means to obtain its result.

⟨*named fun classifications*⟩+≡
```
    Named__(GENERAL,2,IsConvertible,SOMA,isConvertible,ANY,,,,,)
```
Its objective is to determine whether its first argument type is (implicitly) convertible to the second, for which it `provide`s an expression of the first argument type and tries to `accept` it as a function argument of the second argument type.

⟨`IsConvertible`⟩≡
```
    template <>
    struct Named<GENERAL,2ul,IsConvertibleTag>
    {
        template <typename Param1_, typename Param2_ Fill2__(typename=Void)>
        struct Call
        : Bind<Make<B<(sizeof(accept<Param1_>(provide<Param2_>()))==sizeof(Yes))> >,
               Exp<Param1_>,Exp<Param2_> >
        {};
    };
```

`Common`

⟨*named fun classifications*⟩+≡
```
    Named__(GENERAL,2,Common,META,common,ANY,,,,,)
```
The main purpose of this fun is to determine the result type of `Cond` if called with soma arguments as given in § 9.4.2, which is done analogously to the description in [ISO98] § 5.16:

Rvalue arguments of a binary arithmetic operation may, after their promotion, be *converted* implicitly to a wider type according to [ISO98] § 5;9. To obtain this common type for which the operation is to be evaluated, the binary mapping `Common` refers to the trait `ToPriority` which provides an ordering of arithmetic types by their range width just for this purpose. The (meta natural value) result of this trait mapping extends from `N<0>` for `signed int` to `N<6>` for `long double`—more narrow types are not considered, as these are subject to promotion. The type `signed long` gets the priority `N<3>` if it can represent all `unsigned int` values, `N<2>` if not. This is relevant for the case of a `signed long` and an `unsigned int` operand (which has the priority `N<1>`).

If not both arguments have arithmetic or enum type, a simplified implementation just tries to convert either argument to the other.

⟨`Common`⟩≡
```
    template <>
    struct Named<GENERAL,2ul,CommonTag>
    : Cond<BoolAnd<BoolOr<IsArithmetic<P<1> >,IsEnum<P<1> > >,
                   BoolOr<IsArithmetic<P<2> >,IsEnum<P<2> > > >,
           Bind<Cond<IsGreater<ToPriority<P<1> >,ToPriority<P<2> > >,
                     Cond<BoolAnd<IsEqual<ToPriority<P<1> >,N<2> >,
                                  IsEqual<ToPriority<P<2> >,N<1> > >,
                          Make<Exp<unsigned long> >,
                          P<1> >,
                     Cond<BoolAnd<IsEqual<ToPriority<P<1> >,N<1> >,
                                  IsEqual<ToPriority<P<2> >,N<2> > >,
                          Make<Exp<unsigned long> >,
                          P<2> > >,
                ToPromoted<P<1> >,
                ToPromoted<P<2> > >,
           Cond<IsConvertible<P<2>,P<1> >,
                P<1>,
                Cond<IsConvertible<P<1>,P<2> >,
                     P<2>,
                     Bind<Exp<UndefinedFun>,P<1>,P<2> > > > >::Result
    {};
```
This example impressively demonstrates the power of the presented approach, allowing to implement complex type operations in a format that is quite easy to grasp thanks to its functional notation.

### 11.1.3 Object mappings

⟨*general object mappings*⟩≡
    ⟨*qualification*⟩
    ⟨`Address`⟩

The semantics of general object mappings is independent of the semantics of their argument types. This applies only to a rather limited set of mappings, which are of quite common applicability, though.

## Qualification

⟨*qualification*⟩≡

```
    ⟨QualifyConst / QualifyVolatile / QualifyConstVolatile⟩
    ⟨Qualify⟩
    ⟨QualifyLike⟩
```

Several mappings serve to qualify their argument by any of the well-known cv-qualifiers. Note that qualifiers are only appended, but not removed, so that the mapping performs only valid conversions to a more qualified type. There is no mapping to strip off qualifiers of objects, while in § 11.1.1 mappings were provided to do so with types. For reference arguments, the qualification applies to the referenced memory location.

⟨*named fun classifications*⟩+≡

```
    Named__(GENERAL,1,QualifyConst,SOMA,qualifyConst,ANY,,,,,)
    Named__(GENERAL,1,QualifyVolatile,SOMA,qualifyVolatile,ANY,,,,,)
    Named__(GENERAL,1,QualifyConstVolatile,SOMA,qualifyConstVolatile,ANY,,,,,)
```

Those mappings which apply a particular qualifier are implemented in the same way, which is therefore provided by a macro.

⟨QualifyConst / QualifyVolatile / QualifyConstVolatile⟩≡

```
    #define Qualify__(Qual__)                                             \
    template <>                                                           \
    struct Named<GENERAL,1ul,Qualify##Qual__##Tag>                        \
    : Create<To##Qual__<P<1> > >                                          \
    {};

    Qualify__(Const)
    Qualify__(Volatile)
    Qualify__(ConstVolatile)

    #undef Qualify__
```

The ternary mapping `Qualify` takes as its second and third arguments meta boolean values that indicate whether a `const`- or `volatile`-qualification is to be applied to the first argument. If both are `False` objects, the mapping is just the identity.

⟨*named fun classifications*⟩+≡

```
    Named__(GENERAL,3,Qualify,SOMA,qualify,ANY,,,,,)
```

⟨Qualify⟩≡

```
    template <>
    struct Named<GENERAL,3ul,QualifyTag>
    : Cond<P<2>,Cond<P<3>,QualifyConstVolatile<P<1> >,
                          QualifyConst<P<1> > >,
                Cond<P<3>,QualifyVolatile<P<1> >,
                          P<1> > >
    {};
```

`QualifyLike` is a binary variant of the previous mapping.

⟨*named fun classifications*⟩+≡
```
    Named__(GENERAL,2,QualifyLike,SOMA,qualifyLike,ANY,,,,,)
```

The qualification of the second argument is applied to the first, to that the set of qualifiers of the returned object is the union of those of the arguments.

⟨`QualifyLike`⟩≡
```
    template <>
    struct Named<GENERAL,2ul,QualifyLikeTag>
    : Qualify<P<1>,IsConst<P<2> >,IsVolatile<P<2> > >
    {};
```

`Address`

In contrast to the type mapping `ToPointer`, this mapping is bound transparently with a reference argument, returning a pointer to its memory location.

⟨*named fun classifications*⟩+≡
```
    Named__(GENERAL,1,Address,SOMA,address,ANY,
            Prefix,DIRECT,&,DELAYED,Lvalue)
```

While the function version `address` of this fun is defined for all kinds of arguments, the operator version `&` is only provided explicitly for delayed arguments, as its non-delayed application is just the builtin effect, [ISO98] § 5.3;2.

⟨`Address`⟩≡
```
    template <>
    struct Named<GENERAL,1ul,AddressTag>
    {
        template <typename Param1_>
        struct Call
        {
            typedef typename ToPointer<Exp<Param1_> >::Result Result;
            static Result result(Param1_ param1)
            {
                return &param1;
            }
        };
    };
```

### 11.1.4 Function pointers

⟨*implementation*⟩+≡
    ⟨⟨`Dispatch` *for 1ary function pointer call*⟩⟩
    ⟨`Dispatch` *for 2ary function pointer call*⟩
    ⟨⟨`Dispatch` *for 3ary function pointer call*⟩⟩
    ⟨⟨`Dispatch` *for 4ary function pointer call*⟩⟩

Function pointers are a particular case of general semantics mappings. `Dispatch` is directly specialized to accept these, which allows to use them as 0th argument to `bind` and therefore to integrate them into the presented expression system, including to delay their binding and evaluation.

The implementation is basically straightforward, but performs static argument type checking to provide a-priori information whether a call of the function pointed to is legal and to return `Undefined` otherwise.

⟨`Dispatch` *for 2ary function pointer call*⟩≡
```
    template<Level LEVEL_, typename Result_,
             typename Arg1_, typename Arg2_,
             typename Stripped1_, typename Stripped2_>
    struct Dispatch<LEVEL_,true,false,
                    Result_ (*)(Arg1_,Arg2_),
                    Stripped1_,Stripped2_ Fill2__(Void)>
    {
        struct Result
        {
            template <typename Param0_,
                      typename Param1_, typename Param2_ Fill2__(typename)>
            struct Apply
            {
                typedef typename
                        Bind<Cond<BoolAnd<IsConvertible<P<1>,Exp<Arg1_> >,
                                          IsConvertible<P<2>,Exp<Arg2_> > >,
                                  Make<Exp<Result_> >,
                                  Bind<Exp<UndefinedFun>,P<1>,P<2> > >,
                             Exp<Param1_>,Exp<Param2_> >::Result
                        Result;
                static Result result(Param0_ param0,
                                     Param1_ param1, Param2_ param2 Fill2__(Void))
                {
                    return param0(param1,param2);
                }
            };
        };
    };
```

## 11.2 Individual semantics

Individual mappings form the majority of the set of mappings, increasing by any new application domain. While that small subset of them represented by operators is forced by the language to exist, the common case is that of a named fun introduced by some developer, who is responsible of providing a reasonable specification of its semantics.

For instance, the operator `<<` has the meaning of shifting bits (according to some binary representation) to the left for arguments that represent integral values, but was suggested by the developers of C++ and accepted by its users to describe output if the left argument represents a stream. Both interpretations are feasible, while the former has the right of way, being defined by the language for builtin types without a chance to be overwritten (which is different, e.g., in Haskell).

But what happens if the left operand is a bit stream and the right operand a natural value? Shall a bit-representation of this value be shifted into the stream from right to left or as many zeroes as it indicates? Counterexamples like this may seem arbitrary, but appear at unpleasant frequency, the more abstract and broadly applicable a software is intended to be.

Therefore, one should:

- uniquely match individual mapping names and semantics, and
- particularly do so for operators.

To put it shorter: do not overload in terms of semantics! While overloading provides many neat effects for conventionally designed software, it interferes with the approach to generally abstract any function to basically accept any argument type. Either explicit binding or strict semantics can be omitted by genericity, but not both.

These are just recommendations, though, as it is not intended to provide a collection of individual mapping specifications in this document, but just to demonstrate a way to do so. Therefore, only the structure of such specifications is indicated here, not their particular implementations. The specification of an individual mapping consists of two or three parts:

- the *registration* of the mapping by providing an according call of the macro `Named__`,
- a set of alternative general definitions, given by the according `FunExpStackExp` instance, and
- if the mapping is represented by an operator, the according instance of `OperatorTraits` providing validity and result type information for builtin types.

As an example, consider the mapping `IsLess` representing the operator `<`.

⟨*named fun classifications*⟩+≡
```
    Named__(INDIVIDUAL,2,IsLess,SOMA,isLess,ANY,
            Infix,DIRECT,<,ANY,Rvalue)
```

It may be defined by either applying `IsGreater` to its permutated arguments or by negating `IsGreaterEqual` applied to the arguments. These definitions require that those two maps are not implemented in terms of `IsLess`, as this would lead to circular dependencies that could not be resolved and would finally cause the compiler to fail due to templates being nested too deeply.

⟨*implementation*⟩+≡
```
    template <>
    struct FunExpStackExp<IsLessTag>
    : MakeStack<IsGreater<P<2>,P<1> >,
                BoolNot<IsGreaterEqual<P<1>,P<2> > > >
    {};
```

Comparing this to alternative implementations by conventional coding, one should notice the impressive simplicity and power of the presented way of defining mappings.

[ISO98] § 5.9 indicates when and how the operator is applicable to builtin types, which is either to arithmetic or enumeration type arguments or to pointer type arguments, always returning a `bool` result.

⟨`OperatorTraits`⟩+≡
```
    template <typename Param1_, typename Param2_>
    struct OperatorTraits<IsLessTag,Param1_,Param2_>
    {
        template <typename=Void>
        struct IsDefined
        : BoolOr<BoolAnd<BoolOr<IsArithmetic<Exp<Param1_> >,
                                IsEnum<Exp<Param1_> > >,
                         BoolOr<IsArithmetic<Exp<Param2_> >,
                                IsEnum<Exp<Param2_> > > >,
                 BoolAnd<IsPointer<Exp<Param1_> >,
                         IsPointer<Exp<Param2_> > > >
        {};
        template <typename=Void>
        struct ToResult
        : Exp<bool>
        {};
    };
```
It is just a matter of efforts to provide according definitions for the other operators.

# Chapter 12

# Conclusion

## 12.1 Motivation

Performance-critical software relies on semantically identical mapping implementations by different algorithms which are adapted to the data structures to operate on. While polymorphism basically allows to avoid the problem of code replication, common approaches of realization lack either run-time efficiency (dynamic polymorphism) or control of formal correctness by some means of constraining (static polymorphism).

Popular solution strategies for the latter employ separate constructs like concepts to provide additional information on algorithm applicability. To express the requirements appropriately, these have to reflect the described code consistently, which may be accomplished by aggregating them accordingly or by allowing to express their fulfillment by a predicate that can be made part of a boolean requirement expression. Anyhow, this puts more burden on the programmer to keep this additional information consistent with the code, which is aggravated by the fact that different mapping properties have to be reflected by independent constructs, like traits providing associated types and constants. These also have to be kept in accordance with code extensions. Similarly, programming techniques like meta programming and delayed binding, that promise solutions for common coding problems, are defined in a stand-alone manner and require their consideration within each mapping definition as well, which is tedious to maintain.

Another problem is caused by the fact that most of the properties are not provided explicitly or not in a systematic way, as would be necessary to allow their consistent use as objects of meta programs. This forces the user to rely on language capabilities or to hope for language extensions that provide such information or that directly perform some desired effect, rather than to allow to introduce such functionalities by oneself.

## 12.2 Resume

While questions of data structure design were not discussed within this approach, a categorization of types was introduced to support a generic evaluation of mappings, considering different behavioral classes of these and of their arguments. A type category expresses type properties that are independent of a type's individual semantics and describe its role within the evaluation process. It should be noted that this category system has not been claimed to be complete; the techniques developed to obtain and to process category information could be extended without changing the overall structure.

The most relevant of the presented establishings is that of a common, generalized representation of mapping instanciations. It provides in an explicit manner according properties, like the result type as a mapping's effect at the meta stage, or its implementation by a function at the soma stage. The set of properties can be extended by updating a considerably small amount of elementary mapping implementations. The provision by a common construct allows to automatically deduce the properties of a composed mapping from the properties of the participating mappings.

In particular, meta information on the applicability of a mapping to a given tuple of arguments allows to explicitly check the formal correctness of such an expression, which provides the effect of *lazy typing* by meta predicates. It is used to introduce a mechanism to automatically select an appropriate implementation from a list of candidates, which is sequentially applied to the according individual and general definitions of an algorithm. Semantical correctness is enforced if avoiding the semantical overloading of mapping identifiers or operators and assigning each an unique meaning.

The evaluation of mapping instantiations is performed following a fixed scheme that respects certain mapping properties which are independent of the mappings' particular semantics and which refine the general categorization of types. These properties are registered with the according mapping identifiers and eventual operators by simple macro calls, which are invoked whereever category-dependent steps of the evaluation process are defined and save the implementor from reproducing these.

Quite a large part of the implementation is dedicated to the definition of a basic set of mappings, which serve as atomic components of more complex definitions that do not require a hand-crafted implementation anymore, but that are just composed of the former. This is not only a technical and illustrative issue, but also justifies the initial categorization of mappings and proposes a general pattern of providing generic implementations for certain mapping categories.

## 12.3 Evaluation

The presented approach provides a new programming strategy which combines benefits of object-oriented, generic and meta programming as being modular, flexible and expressive at (theoretically) optimal efficiency. The multistage formulation provides reusability not only across variations of data structures and algorithms, but even at different stages of compilation.

At the same time, the functional programming style was followed and powerful constructs characteristic of that paradigm were introduced which base on the concept of delayed binding. At small and fixed amount of structural overhead, it allows the elegant and expressive definition of mappings by mere compositions. Analogous definitions by conventional code were possible, but would be dominated by far by technical issues to be governed by the programmer. According properties are provided automatically, as are techniques like meta programming and delayed binding. These features are illustrated by the implementation of the approach itself, which employs its own constructs at considerable complexity wherever this does not lead to circular dependencies.

The programmer has a great benefit from these advantages: She can implement code in a very compact, functional manner, only expressing the algorithmic aspect of the implementation. Technical issues are encapsulated and do not have to be considered anymore, which means that coding is much easier than in plain C++, while language details and programming techniques are automatically respected. Although one has to get accustomed to the particular syntax of those constructs relevant for the application programmer, their range is very narrow compared to the set of syntactical issues related to common functions, which promises a short learning period.

At the same time, the code is much more powerful than in classic C++ and other programming languages: The inherent meta programming facilities automatically cause constant computations to be performed at compile time without any need of particular code adaptions. Types and objects are processed by mappings of equal syntax, which may be mixed. Delayed binding extends the expressiveness of the language to allowing the implicit definition of mappings at the place of their use, which dramatically simplifies the incorporation of user-defined algorithms, e.g., the ad-hoc-specification of a mapping representing a (mathematical) function to be passed to an integration algorithm.

Static reflection, i.e., the explicit availability of mapping properties at the meta stage, allows to solve different substantial problems of both technical and design matter. By the explicit selection of a mapping's implementation, free binding is supported while keeping control of the validity of expressions. This overcomes different restrictions of the commonly applied techniques for realizing constrained genericity to answer the classic question of how to associate multiple data structure and algorithm implementations. It is driven by the technical novelty of a case-dependent explicit choice of an algorithm's implementation, stepping back from imposing formal requirements on data structures and algorithms to have the compiler select the appropriate bindings.

The common evaluation scheme allows to realize multiple dispatches by consistent argument resolution and the conditional, automatically performed choice of a defined implementation. On the one hand, the approach is more strict than common methods, requiring a named mapping to have fixed, immutable semantics. On the other, it allows to provide a mapping implementation independently of its alternatives, eliminating any burden of naming strategies and of understanding and exploiting language rules of matching definition specializations. In any case, one should be aware that there exists no alternative approach of solving the given problems in a more satisfactory way.

As a consequence, not just the syntax, but also the association of an algorithm's implementation becomes much simpler: By default, mapping implementations are provided globally. Only if directly supported by member data, a specialized implementation is provided within the definition of a data structure.

Furthermore, the approach provides a precise place to eventually apply more intelligent and powerful mechanisms for selecting a mapping implementation, which would not just make a static choice based on validity, but which might, for instance, respect case-dependent properties like performance issues. Meanwhile, performance conditions may be flexibly enforced by introducing them as explicit predicates.

In fact, many of the techniques shift the responsibility of code creation from the implementor to the compiler. This takes a lot of burden off the former, as many C++ constructs were encapsulated and do not have to be considered and repeated anymore, even not to be understood.

An example of different kind are the preprocessing stage code constructs used for meta and soma stage code generation by the automatized definition of the mapping evaluation process based on the categorization of mappings. This limits their implementation to that of the effective application, without needs of reproduction for future mapping definitions.

Besides introducing those individual achievements, this presentation illustrates the power of abstraction without performance overhead, as provided by generic programming. By its consequent application, many programming problems can be solved (unless inherent to the application domain), usually resulting in more elegant and effective structures instead of tedious workarounds.

Vice versa, if criticizing the presented techniques due to their abstractness, unfamiliarity or experimental nature, one has to be aware that each step in their development was driven by some actual problem to be overcome. The intent of avoiding certain constructs in favor of holding on to more conservative programming methods is necessarily paid for by the well-known effects of those problems. The executable code created from source code expressed by means of the presented constructs could be generated as well from conventional code if written appropriately; there is no secret language detail (yet?) which would allow to produce better machine code by those techniques. But there are significant benefits from the higher structural level of the new approach, which will pay off any time the code is inspected, modified or extended. The expressive notation and the possibility of introducing optimizations locally, while obtaining their effect globally due to the common evaluation process, allow to quickly develop highly efficient yet modular code. Their analogous handwritten implementation would become unmanageable from a certain point of complexity on, as the amount of technical matters to respect would be too confusing to the human reader and programmer, as may be verified by looking at the source code of actual generic software projects.

On the other hand, besides the static evaluation of the participating constructs being advantageous to run-time performance, the structural advantages of the presented approach recommend its application to any kind of problem implemented in terms of a significant variety of interacting algorithms and data structures. The design drawbacks of inheritance hierarchies and of other approaches to constraining genericity show up in most examples of non-trivial structure. While compilation efficiency forbids to make this method a general way of programming, the core parts of many applications would profit from being designed in the presented, component-based manner.

It must be noted that the presented approach depends on C++, but might be translated to other languages providing unconstrained genericity and means of specialization. It is rather a design method than a particular C++ framework.

## 12.4 Outlook

This work should most naturally be continued by providing a considerable range of applications, both of mappings directly implemented by the presented methods and of data structures implemented in terms of those. This has to be preceded by a careful discussion of the according problem domain, in order to introduce meaningful definitions of each entity's semantics. One finds today's discussions dominated by these topics rather than by technical issues.

The suggested methodology differs from well-established approaches, e.g., using concepts, in particular for the type semantics being fixed and for any mapping being free and its validity to be verified explicitly. This allows to develop alternative formulations for established applications like those of the STL, which due to the improved expressiveness and to stricter abstraction of this approach promises to result in solutions that are more widely applicable and that model their domain more closely.

The definition of data structures itself induces a large amount of new questions, leading to a similar demand for a common method of implementation answering those questions generically. In fact, the presented approach was originally motivated by the author's intense research on such questions, which led to the conclusion that their solution would have to rely on a systematic way of defining algorithms to provide some of the mentioned features. After having proposed an according approach in this text, the original task of developing a data structure design method may return into focus.

But various questions and problems within the presented approach have not been answered yet, either. Among them, a very interesting one is the explicit representation of the complexity of a mapping evaluation, which could be applied for the automatical selection of the most efficient implementation from a set of alternatives, and which would then allow the implemention of further optimizations like partial or parallel evaluation, exploiting the locality of common structures responsible for evaluating mappings.

Finally, the notation of several of the presented constructs suffers from particularities or even deficiencies of C++, while many other language features are not addressed at all. Therefore, it seems both feasible and recommendable to define a language extension that uses more consistent terms. These could be translated to C++ code by some preprocessor, guaranteeing the code's portability and quality of optimization. On the other hand, it might turn out beneficial to examine and tune the way the compiler processes the constructs which occur most frequently. Due to the extreme and non-conventional use of template classes for these purposes, its performance and therefore the techniques' applicability might be improved substantially by true extensions of the language to directly support such applications.

# Bibliography

[AAD$^+$05] David Abrahams, J. Stephen Adamczyk, Peter Dimov, Howard E. Hinnant, and Andreas Hommel. A proposal to add an rvalue reference to the C++ language. Technical report, ISO C++ working group, 2005. Doc. No. ANSI X3J16-05-0030 / ISO WG21-1770
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1770.pdf`.

[AG05] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming*. Pearson Education, Inc., 2005.

[Ale01] Andrei Alexandrescu. *Modern C++ Design*. C++ In–Depth Series. Addison Wesley Longman Publishing Co., Inc., 2001.

[Aus98] Matthew H. Austern. *Generic Programming and the STL*. Addison Wesley Longman Publishing Co., Inc., 1998.

[Ber00] Guntram Berti. *Generic software components for scientific computing*. PhD thesis, Faculty of mathematics, computer science, and natural science, BTU Cottbus, Germany, 2000.

[Boo] Boost homepage.
`http://www.boost.org`.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison Wesley Longman Publishing Co., Inc., 2000.

[DH96] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in C++. In *Proceedings of the eleventh annual conference on Object-oriented programming systems, languages, and applications*, pages 306–323. ACM Press, 1996.

[GJK$^+$05] Douglas Gregor, Jaakko Järvi, Mayuresh Kulkarni, Andrew Lumsdaine, David Musser, and Sibylle Schupp. Generic programming and high-performance libraries. *International Journal of Parallel Programming*, 33(2), June 2005. To appear.

[GLA] Generic Linear Algebra Software project (GLAS) homepage.
`http://glas.sourceforge.net`.

[GSW$^+$05] Douglas Gregor, Jeremy Siek, Jeremiah Willcock, Jaakko Järvi, Ronald Garcia, and Andrew Lumsdaine. Concepts for C++0x (revision 1). Technical Report N1849=05-0109, ISO C++ working group, August 2005.

[HAD04] Howard E. Hinnant, David Abrahams, and Peter Dimov. A proposal to add an rvalue reference to the C++ language. Technical report, ISO C++ working group, 2004. Doc. No. ANSI X3J16-04-0130 / ISO WG21-1690
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1690.pdf`.

[HDA02] Howard E. Hinnant, Peter Dimov, and David Abrahams. A proposal to add move semantics support to the C++ language. Technical report, ISO C++ working group, 2002. Doc. No. ANSI X3J16-02-0035 / ISO WG21-1377
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.pdf`.

[ISO] ISO/IEC JTC1/SC22/WG21 C++ working group homepage.
`http://www.open-std.org/jtc1/sc22/wg21/`.

[ISO98] ISO C++ working group, Washington D.C. *ISO/IEC 14882 International Standard for Information Systems – Programming Languages – C++*, 1998.

[Jär99] Jaakko Järvi. Tuples and multiple return values in C++. Technical Report 249, Turku Centre for Computer Science, March 1999.

[JLSW03] Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An analysis of constrained polymorphism for generic programming. In Kei Davis and Jörg Striegnitz, editors, *Multiparadigm Programming in Object-Oriented Languages Workshop (MPOOL) at OOPSLA*, Anaheim, CA, October 2003.

[Jos99] Nicolai M. Josuttis. *The C++ Standard Library.* Addison Wesley Longman Publishing Co., Inc., 1999.

[Joy99] Ian Joyner. *Objects Unencapsulated: Java, Eiffel, and C++.* PTR. Prentice Hall, 1999.

[JP] Jaakko Järvi and Gary Powell. Boost Lambda Library (BLL) homepage.
`http://www.boost.org/libs/lambda`.

[JS04] Jaakko Järvi and Bjarne Stroustrup. Decltype and auto (revision 3). Technical report, ISO C++ working group, 2004. Doc. No. ANSI X3J16-04-0047 / ISO WG21-1607
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1607.pdf`.

[Kli01] Nils Klimanis. Lösungsverfahren und Vorkonditionieren in objekt-orientierter Implementierung für Konvektions-Diffusions-Reaktions-Probleme. Master's thesis, Institute for Numerical and Applied Mathematics, University of Göttingen, 2001.

[Koh00] Joachim Kohlhammer. Erzeugung und Verfeinerung von Finite-Element-Triangulierungen. Master's thesis, Institute for Numerical and Applied Mathematics, University of Göttingen, 2000.

[Koz05] Bronek Kozicki. A proposal to add l-value member function qualifier. Technical report, ISO C++ working group, 2005. Doc. No. ANSI X3J16-05-0044 / ISO WG21-1784
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1784.pdf`.

[Mad] John Maddock. Boost type traits homepage.
`http://www.boost.org/libs/type_traits/`.

[MC00] John Maddock and Steve Cleary. C++ type traits. *Dr. Dobb's Journal*, October 2000.

[Mey95] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison Wesley Longman Publishing Co., Inc., 1995.

[Mey97] Scott Meyers. *Effective C++: 50 specific ways to improve your programs and designs*. Addison Wesley Longman Publishing Co., Inc., 2nd edition, 1997.

[MK] Paul Mensonides and Vesa Karvonen. Boost preprocessor library homepage.
`http://www.boost.org/libs/preprocessor`.

[Moh] B. Mohr. WWW C++ information.
`http://www.kfa-juelich.de/zam/cxx/extern.html`.

[Mye95] Nathan Myers. A new and useful template technique: "Traits". *C++ Report*, 7(5):32–35, June 1995.

[Par02] Michael Pargmann. Adaptive Gitterverfeinerung in drei Dimensionen. Master's thesis, Institute for Numerical and Applied Mathematics, University of Göttingen, 2002.

[Pri96] Andreas P. Priesnitz. Untersuchung iterativer Lösungsverfahren am Beispiel diskretisierter Konvektions-Diffusions-Reaktions-Gleichungen. Master's thesis, Institute for Numerical and Applied Mathematics, University of Göttingen, 1996.

[Pri01] Andreas P. Priesnitz. Generative container implementation in C++. In *Third International Symposium on Generative and Component-Based Software Engineering (GCSE) – Young Researchers Workshop*, September 2001.

[Pri03] Andreas P. Priesnitz. C++ techniques for scientific programming. Internal Report, Institute for Numerical and Applied Mathematics, University of Göttingen, 2003.

[Ram] Norman Ramsey. Noweb homepage.
`http://www.eecs.harvard.edu/~nr/noweb/`.

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, September 1994.

[Rei04] Benjamin Reichardt. Objektorientierte Kapselung paralleler Kommunikation für Clusterumgebungen. Master's thesis, Institute for Numerical and Applied Mathematics, University of Göttingen, 2004.

[RHC$^+$96] John V. W. Reynders, Paul J. Hinker, Julian C. Cummings, Susan R. Atlas, Subhankar Banerjee, William F. Humphrey, Steve R. Karmesin, Katarzyna Keahey, Marikani Srikant, and Mary Dell Tholburn. POOMA: A Framework for Scientific Simulations of Paralllel Architectures. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming in C++*, chapter 14, pages 547–588. MIT Press, 1996.

[Sch00] Heiko Schilling. Flußorientierte Numerierungsstrategien zur Vorkonditionierung konvektionsdominanter Probleme. Master's thesis, Institute for Numerical and Applied Mathematics, University of Göttingen, 2000.

[Sie] Jeremy Siek. Matrix Template Library (MTL) homepage.
`http://www.osl.iu.edu/research/mtl`.

[Sie99] Jeremy G. Siek. A modern framework for portable high performance numerical linear algebra. Master's thesis, University of Notre Dame, 1999.

[SL95] Alexander A. Stepanov and Meng Lee. *The Standard Template Library*. Hewlett Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, February 1995.

[SL98] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A unifying framework for numerical linear algebra. In *ECOOP Workshops*, pages 466–467, 1998.

[SL00] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming*, Erfurt, Germany, October 2000.

[Sma] Yannis Smaragdakis. FC++ homepage.
`http://www.cc.gatech.edu/~yannis/fc++`.

[SPE] SPEC homepage.
`http://www.spec.org`.

[SR03] Bjarne Stroustrup and Gabriel Dos Reis. Template aliases for C++. Technical report, ISO C++ working group, 2003. Doc. No. ANSI X3J16-03-0072 / ISO WG21-1489
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1489.pdf`.

[SR05] Bjarne Stroustrup and Gabriel Dos Reis. A concept design (rev. 1). Technical report, ISO C++ working group, 2005. Doc. No. ANSI X3J16-05-0042 / ISO WG21-1782
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1782.pdf`.

[Str] Jörg Striegnitz. FACT! homepage.
`http://www.kfa-juelich.de/zam/FACT`.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley Longman Publishing Co., Inc., 4th edition, 2000.

[Sut00] Herb Sutter. *Exceptional C++: 47 engineering puzzles, programming problems, and solutions*. Addison Wesley Longman Publishing Co., Inc., 2000.

[Sut02] Herb Sutter. *More exceptional C++: 40 new engineering puzzles, programming problems, and solutions*. Addison Wesley Longman Publishing Co., Inc., 2002.

[Taha] Walid Taha. MetaOCaml homepage.
`http://www.metaocaml.org`.

[Tahb] Walid Taha. Multi-stage programming homepage.
`http://www.cs.rice.edu/~taha/MSP`.

[Vela] Todd L. Veldhuizen. Blitz++ homepage.
`http://www.oonumerics.org/blitz`.

[Velb] Todd L. Veldhuizen. The object-oriented numerics page.
`http://www.oonumerics.org/oon`.

[Vel95a] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[Vel95b] Todd L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[Vel99] Todd L. Veldhuizen. Techniques for scientific C++. Technical report, Indiana University Computer Science, 1999.

[Vel01] Todd L. Veldhuizen. *Blitz++ User's Guide*, 2001.
`http://www.oonumerics.org/blitz/manual/blitz.html`.

[VJ03] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates – the Complete Guide*. Addison Wesley Longman Publishing Co., Inc., 2003.

[WK] Joerg Walter and Mathias Koch. Boost Basic Linear Algebra Library (uBLAS) homepage.
`http://www.boost.org/libs/numeric/ublas/`.

[WSL01] Jeremiah Willcock, Jeremy Siek, and Andrew Lumsdaine. Caramel: A concept representation system for generic programming. In *Second Workshop on C++ Template Programming*, Tampa,Florida, October 2001.

# List of examples

# List of figures

# List of tables

# Index

# Acknowledgements

Ich danke Herrn Prof. Dr. G. Lube und dem Institut für Numerische und Angewandte Mathematik der Universität Göttingen für die Ermöglichung der Durchführung meines Promotionsvorhabens, nicht zuletzt durch die Gelegenheit, dieses für längere Zeit durch Tätigkeiten in Forschung, Lehre und Verwaltung finanzieren und dabei viele interessante Erfahrungen sammeln zu können. Im gleichen Sinn danke ich der Prof. Schumann GmbH in Göttingen für die meinen persönlichen Zielen entgegenkommende Gestaltung meines gegenwärtigen Arbeitsverhältnisses.

Insbesondere danke ich Herrn Prof. Dr. G. Lube und Herrn Prof. Dr. R. Schaback dafür, die Aufgabe des Referats respektive Korreferats für die vorliegende Arbeit übernommen zu haben.

Die vergangenen Jahre gaben mir ausreichend Anlaß, meiner Familie und vielen feinen Freunden (die sich bitte in diesem kollektiven Ausdruck nicht als einzelne übergangen zu sein verstehen) für Geduld, Verständnis, Rückhalt und Ermunterung von Herzen dankbar zu sein, welche sie mir in unerwartet hohem Maße entgegenbrachten, auch wenn ich dies viel zu oft nicht entsprechend erwidert habe. Ihr habt mir sehr geholfen!

Namentlich möchte ich an dieser Stelle jene nennen, denen ich Anregung und Unterstützung im direkten Zusammenhang mit der Erstellung dieser Arbeit verdanke. Außer von Heiko Schilling, Nils Klimanis, Joachim Kohlhammer, Emmanouil Stafilarakis, Michael Pargmann und, nur zeitlich zuletzt, Benjamin Reichardt, die mir im Rahmen unserer gemeinsamen Bestrebungen eine thematisch und menschlich inspirierende Seilschaft boten, erhielt ich erhebliche Motivation fachlicher, persönlicher oder strategischer Natur aus Gesprächen mit Markus Rütten, Jodel, Rüdiger Siebert, Cliff Seidlitz, Annet Göhmann-Ebel, Carsten Damm, Ulrich Weihofen, Nicola Botta, Cezar Ionescu, Rupert Klein, Sibylle Schupp, Guntram Berti, Krysztof Czarnecki und so manchen anderen, die mir ihre Nichterwähnung verzeihen mögen.

Und im tiefen Bewußtsein der Abhängigkeit meines Wirkens von praktischen Gegebenheiten danke ich dem guten Tio Dirk sowie Petra Trapp, Rolf Wassmann, Joachim Perske und Gerhard Siebrasse für immer freundlichen Beistand beim Kampf gegen Technik, Bürokratie und die eigenen Unzulänglichkeiten.

# Curriculum Vitae

Ich wurde am 12. Mai 1972 in der Universitätsstadt Göttingen als deutscher Staatsbürger geboren und widmete meine späteren Bemühungen unter anderem dem Besuch des Eichsfeld-Gymnasiums in Duderstadt, wofür ich im Mai 1990 mit dem Abitur ausgezeichnet wurde.

Hierdurch angespornt, begann ich im Oktober 1990 mit dem Studium der Mathematik an der Georg-August-Universität Göttingen, unterbrochen durch einen Aufenthalt an der University of California in San Diego, USA, im Studienjahr 1992/93. Im Juni 1992 wurde mir ein Vordiplom, im Oktober 1996 ein Diplom im Fach Mathematik verliehen. Meine unter Betreuung durch Prof. Dr. G. Lube verfaßte Diplomarbeit hatte den Titel *Untersuchung iterativer Lösungsverfahren am Beispiel diskretisierter Konvektions-Diffusions-Reaktions-Gleichungen*

Nach dem Absolvieren eines freiwilligen sozialen Dienstes in San Felipe, Chile, anstelle eines Zivildienstes begann ich im Oktober 1998 mein Promotionsvorhaben in der Arbeitsgruppe von Prof. Dr. G. Lube am Institut für Numerische und Angewandte Mathematik der Universität Göttingen. Dieses stand bis Februar 2004 in Verbindung mit einer Anstellung für Aufgaben in Forschung, Lehre und Verwaltung und mündete nun in dem vorliegenden Werk.