The thesis consists of an introduction and 6 reprints of papers published earlier.

New Techniques in Generic Programming –

C++ is more Intentional than Intended

Jaakko Järvi

To be presented, with the permission of the Faculty of Mathematics and Natural Sciences of the University of Turku, for public criticism in the Auditorium of the Computer Science Department on June 9th, 2000, at 12 noon.

University of Turku
Department of Computer Science
Turku, Finland
2000

ISBN 951-29-1721-1
ISSN 1239-1883
Painosalama Oy
Turku, Finland

Abstract

Traditionally generic programming is about parameterising containers and algorithms with respect to the types of the elements they contain or operate on. Recent findings have demonstrated that the expressive power of generic programming reaches beyond this. This thesis explores these new capabilities of generic programming; the main emphasis is in the template model of the programming language C++.

The thesis demonstrates that C++ templates form a sublanguage of their own. This static *metalanguage* operates on types and constants, and is interpreted by the compiler. The existence of such a metalanguage is accidental, but it has turned out to be a powerful feature, allowing non-trivial computations to be performed at compile time. Programming with this metalanguage is known as *template metaprogramming*.

The thesis discusses the abstraction mechanisms in common programming languages and points out some of their inadequacies. Particularly, it may be difficult to achieve high intentionality and efficiency simultaneously. The thesis shows that template metaprogramming can be of help, providing more freedom and fine-grained control in the definition of abstractions. With template metaprograms, the programmer can define how abstractions are transformed into compileable code. All this aims at better intentionality of programs.

In a concrete level, the thesis discusses the characteristics of the C++ template model and describes the template metalanguage. It shows how data structures can be defined and manipulated with template metaprograms. As applications, efficient sparse vector and matrix abstractions are presented; the compiler takes care of the bookkeeping of zeroes and nonzeroes. These matrices are further applied in automatic differentiation. The power of the C++ template model and template metaprogramming is demonstrated by template libraries which extend the C++ language itself with tuple types and a mechanism for partial function application.

Acknowledgements

This work was carried out at the Department of Computer Science, University of Turku and at the Turku Centre for Computer Science during the years 1995-2000. I am indebted to many who have helped along the way.

The work started with biomedical signal analysis and gradually shifted closer to software engineering. Not much of the early work with NMR spectra quantification found its way to the final thesis. Nevertheless, I enjoyed working with Samuel Nyman, NMR expert in chemistry, and Markku Komu, physicist in the Turku University Central Hospital as well as with Docent Jari Forsström, who set off the NMR project and arranged financing for the most part of my work. I have had the privilege to work with Jari in many interesting projects, not always directly related to my thesis. I am thankful for his support and confidence; as well as for frequently reminding me to keep the Ph.D. thesis at the highest priority.

I am grateful to my supervisors, Assistant Professor Timo Knuutila and Professor Olli Nevalainen. The first years were easy for you, since I was rather self-supported and didn't bother you much. I guess I have taken that back with interest during the last year or so. Thank you for both the guidance and the freedom to go where my enthusiasm was.

Professors Johan Lilius and Jukka Paakki kindly accepted the task of reviewing the thesis. I believe their feedback helped to improve the thesis and I wish to thank them for their time and effort.

I wish to thank all my colleagues and coworkers in TUCS. Special thanks to Harri Hakonen for many fruitful discussions and for reading and commenting my manuscripts. I would also like to thank Antti Koski, Mauno Rönkkö, Jan-Christian Lehtinen, Timo Raita, Timo Kestilä and Joonas Lehtinen for their comments, opinions and technical help. I would also like to thank Leila Roti and Jouni Smed for spell-checking and correcting the grammar of some of the articles in the thesis.

I also want to thank Gary Powell, Senior Software Engineer at Sierra Ltd. Gary saw the value of my work, and together we have built upon it. With Gary's help, many of the results of the thesis have become very concrete and are available for C++ programmers to be used in every day work. Gary also made me believe in (code) fairies again: It is magical to notice in the morning that new code has been written and bugs have been fixed during the night! Seriously, working with a 9-hour time difference is a big benefit.

My warm thanks are due to the head of the Department of Computer

Science, Professor Timo Järvi, who is my father as well. To save space I choose one, the most concrete and laborious, of the vast amount of reasons to thank him – and my mother: it is not always easy to combine intensive work and family life with small children - and two dogs! Thank you for lessening our load by providing a good home for our dogs Eemu and Valpo.

I want to express my gratitude to Antero Järvi for the help and support he has given as a colleague, as a brother and as a friend.

I am thankful for my friends and relatives for providing me with a life outside work. Most importantly, I want to express my loving thanks to my family. Thank you Leena for your unconditional love and encouragement, and thank you Osmo, Vilja and Viivi for just being there.

This work has been supported by grants from Tekes, the National Technology Agency, from the Academy of Finland and from TUCS.

Contents

Chapter 1

Introduction

Parameterisation is a natural abstraction mechanism for common real-world concepts. For example, the relation between a list of words and a list of numbers is through the common concept of *list*. A list parameterised with the element type is capable of representing both of the above concrete list types, and several others as well. *Generic programming* is the programming methodology for writing parameterised programs or subprograms.

The definition of containers and algorithms parameterised with respect to the types of the elements they contain and operate on is a traditional and typical use of generic programming. Generic container libraries implemented with different programming languages commonly provide abstract data types for lists, vectors, stacks etc., as well as generic procedures for manipulating and traversing the elements of these containers. [SL94, Mey95, MN99]

Recently, the picture of generic programming has started to change. The development in the generic programming facilities of some widespread programming languages, mainly C++, has brought generic programming into spotlight. Parameterisation has replaced class hierarchies and inheritance as the foremost abstraction mechanism in several object oriented class libraries. The container and algorithm library in the standard C++ [C++98] is a prime example of this development, taking traditional generic programming into extremes: the library consists of small parameterised orthogonal pieces of code, which can be assembled together to form concrete subprograms. However, it is possible to go beyond that. Novel programming techniques exploit generic programming in somewhat non-traditional tasks, such as code generation, partial evaluation and static configuration of software elements [Cza98, Eis97, Vel95b, Vel99]. A common denominator in this new line of work is the generative nature of the parameterised definitions. A concrete subprogram is not formed by just filling holes of a fixed skeleton code but literally generated from an abstract description.

This thesis focuses on the recent advances and new possibilities in generic programming, accentuating particularly the generative aspects. The publi-

cations constituting the main body of this thesis propose concrete programming techniques, designs and fully functional generic libraries, which take advantage of the generative capabilities of parameterisation. They provide solutions for numerical programming as well as to programming in general; the last three publications describe generic libraries which, in effect, extend the C++ language with new constructs. The implementation language in the publications is C++ and the results are to some extent specific to this language. This introductory part aims at giving a broader picture of the current research activity in the area of generic programming in general.

1.1 Motivation

Programming is about implementing concepts of some real-world *domain* in a programming language. Ideally, the *domain-specific* real-world concepts can be naturally translated to the implementation language. In such a case, the conceptual distance between the domain-specific and implementation language concepts, known as the *semantic gap*, is small. An implementation language (or environment) with a small semantic gap to the domain language is said to be highly *intentional* [CEG+98]. An implicit requirement for high intentionality is that the abstractions for domain-specific concepts written in the implementation language do not compromise efficiency.

Unfortunately, it is not rare that the implementation language lacks means to express the *domain-specific concepts* in an abstract fashion. As a result, the domain specific representation may be lost in the implementation details and render the implementation difficult to adapt or reason about. As a very simple and concrete example of this problem, consider the language of mathematics as the real-world domain and an implementation language, say Java [AG96], lacking operator overloading capabilities. A domain language expression, such as $x = \frac{-b+\sqrt{b^2-4ac}}{2a}$, where $x, a, b, c \in \mathbb{C}$, looses the intuitiveness in the conversion to the implementation language representation:

```
Complex b2      = Complex.multiply(b,b);
Complex ac4     = Complex.multiply(4,Complex.multiply(a,c));
Complex discrim = Complex.sqrt(Complex.subtract(b2,ac4));
Complex x       = Complex.divide(Complex.subtract(discrim,b)),
                                 Complex.multiply(2,a)));
```

1.1.1 Striving for high intentionality

There is an unbound number of different domains with different concepts, different notations and different abstractions. Consequently, a single programming language cannot support the concepts for every domain directly.

To cope with this diversity, general purpose programming languages allow the *programmer* to define new abstractions and compose these abstractions to produce programs.

The tools for defining abstractions differ between languages. As *object oriented programming* (OOP) is the dominating programming paradigm today, we take a closer look at the most characteristic abstraction mechanisms offered by OOP:

- *Abstract data types and encapsulation:* The programmer can encapsulate a user defined data type and an accompanying set of operations. The implementation of the operations can be separated from their specifications. The encapsulation unit is called a *class* in most object oriented languages.

- *Inheritance:* A class can inherit attributes and operations from another class and redefine or complement some of the inherited operations.

- *Polymorphism and dynamic binding:* Polymorphic operations can be defined to perform different actions for different types of objects. Dynamic binding allows the selection of the operation to be made safely at run time.

This is an impressive set of abstraction tools. Hierarchical real-world concepts can be implemented via inheritance. Inheritance and dynamic binding are crucial for flexible designs and reuse. The usefulness of the abstraction mechanisms is evident, which is demonstrated by the success of OOP. Improvement of understandability, better management of complexity, extensibility, adaptability, reusability and maintainability are identified as some of the main contributions of the OO paradigm [CES97]. However, despite some enthusiastic argumentation [Cox90] it is also evident that OOP has not fulfilled all of its expectations [Web95].

1.1.2 Unsolved problems

Object oriented software is not inherently reusable or adaptable, rather the software must be designed having reusability and adaptability aspects in mind. The *variation points*, that is, the locations for adaptation and reuse, must be explicitly built to the program. *Design patterns* and *frameworks* [GHJ+95] have become extensively popular in software industry and proven to be useful in this respect. They provide sophisticated and well thought designs with clear variation points and allow the reuse of designs for commonly reappearing problems. Ironically, besides being excellent showcases of the power of the object oriented methodology, design patterns also demonstrate many of the problems in OO abstraction mechanisms:

- *Scattered designs:* Design patterns are typically implemented as small code fragments in several collaborating classes. Domain specific concepts are commonly participating at several patterns in different roles. As a result the domain specific concepts, and the patterns themselves, are scattered throughout the code rather than being cleanly encapsulated. This leads to the loss of design information and hinders reuse and adaptation. [Bos98]

- *Performance:* Variation points are typically implemented with dynamic binding as a layer of indirection. This may have a significant negative effect on performance. Another source for performance degradation is that the structure of the compositional abstractions is preserved in the running code, though the structure of an efficient implementation might be totally different. This is particularly apparent in areas involving heavy numeric computations, such as scientific computing and image processing [Han94, MKL97].

- *'Non-object oriented' concepts, aspects:* There are many concepts which do not fit well to the OO view of the world. Such concepts are generally called *aspects* [KLM+94]. Synchronisation, error-handling and performance-related issues are examples of aspects. It is characteristic to aspects that their implementation cannot be encapsulated neatly into a single class or function, rather the implementation spreads to several methods of many classes. This leads to *'tangled'* code, where design decisions are scattered throughout the code.

These problems boil down to a single more general problem: a programming language lacks means to intentionally and efficiently represent certain domain specific concepts. Basically the abstractions in conventional general purpose programming languages support substitution-based compositions: predefined software elements can be assembled into the final application in a fixed pattern but there are no possibilities to alter the internal structure of the assembled software elements or the way they are assembled.

1.1.3 Proposed solutions

There are basically two ways to approach the above problem. The first is to provide a *domain-specific programming language* (DSL) for each domain. The other is to *redesign the abstraction mechanisms* a programming language should provide.

The obvious disadvantage of the first approach is the high cost and difficulty of developing new languages and compilers. Also, using two or more separately developed domain-specific languages in a single application might lead to conflicts. The advantage is the complete freedom in the design of the abstractions for domain specific concepts.

While figuring out how to retain this freedom in the second approach the first approach still seems to be necessary in part; in order to cover the unlimited set of domain specific abstractions the abstraction mechanisms must be somehow generative in nature. The problem is how to avert the drawbacks of the first approach.

Several approaches have been proposed and are in use to provide generative abstraction mechanisms for programming languages and environments. These approaches include:

- *Program generation systems:* configurable systems for implementing domain-specific languages. Such tools can generate compileable programs from specifications written in a predefined DSL. The generation phase may include arbitrary computations, which distinguishes generators from standard compilers. See [Cza98, chapter 6] and [Big97] for summaries of available program generators.

- *Open compilers:* compilation systems with predefined hooks and variation points for modifying and adapting the compiler functionality, e.g for introducing syntactic extensions, optimisations etc. Examples of such systems are Vanilla [DNW+99], Open C++ [Chi95] and DCO [Bos97]. The *Intentional Programming* environment (IP) [Sim95, IP99] developed at Microsoft takes this approach to its extremes, making the whole programming environment (language constructs, debugging, means for editing etc.) configurable and extensible. The IP system is actually a program generator as well.

- *Program weavers:* This item refers mainly to *aspect oriented* programming environments. A special language processor, the aspect weaver, combines the *component language* program, the basic application logic, and a set of *aspect language* programs into a tangled compileable code. The key idea is that even though the code of the final application is complicated and contains a lot of implementation details in a non-localised way, this does not break the encapsulation in the source programs. [KLM+94]

  A related programming paradigm is *literate programming*. Literate programming systems contain a weaver as well. The main focus is, however, in generating a high-quality documentation in addition to the tangled compileable program from a single *literate* program. [Knu92, KL93].

- *Multilevel languages:* programming languages that allow code to have different binding times. A program written in a two-level language can contain static code evaluated at compile-time and dynamic code, which is compiled and executed. Static code can be used for code generation, optimisation etc. The genericity model of C++ makes C++ a two-level

language. Computations with constants and types can be performed during the template instantiation phase of compilation. This corresponds to the static code evaluation. [Vel99].

1.2 Generative programming and active libraries

The line of work just described falls within a newly proposed programming paradigm, *generative programming*, defined as: *'designing and implementing software modules which can be combined to generate specialised and highly optimised systems fulfilling specific requirements* [Eis97]. The common idea can be seen as 'opening up' the compiler to some extent, and also to load some more work to the compiler, or generator. The compiler has to potentially perform some non-trivial computations to map the domain-specific abstractions to executable code.

The spectrum of different implementation technologies for generative programming range from full-scale program generators and open compilers to modest meta-programming capabilities of existing programming languages. While there are demonstrations of the success of true meta-level processing systems, the supporting tools are complex. Adopting such a tool requires long-term commitment [Big97]. On the other end of the spectrum the metaprogramming capabilities of existing programming languages are readily available albeit their expressiveness is restricted. There are a number of successful applications which exploit the metaprogramming facilities of C++, particularly in the field of scientific computing (see section 3.2).

The concept of *active libraries* is closely related to generative programming: they are libraries which implement generative programming ideas. Such libraries extend the compiler by defining domain specific abstractions and the means to transform the abstractions to compileable code. Active libraries may generate components, specialise algorithms, optimise code, tune themselves for a given machine architecture etc. Conceptually they lie somewhere between a compiler and a traditional subroutine or class library. [CEG+98]

1.3 Outline

The preceding discussion describes the general background of the thesis. The rest of the thesis is on a more concrete level, concentrating on a particular aspect of this framework: active libraries and metaprogramming in C++. New metaprogramming techniques and a set of active libraries which utilise the metaprogramming facilities of C++ are described in the six included articles.

Prior to dwelling on the rather detailed and concrete C++ specific articles, the next chapter continues with a discussion of generic programming in

general. The differences and commonalities of the generic programming constructs adopted in some popular programming languages are examined. We focus on the metaprogramming capabilities, and explain why C++ is unique in this sense among the discussed languages.

Chapter 3 discusses C++ templates as a metaprogramming tool and introduces programming techniques typical for metaprogramming. Some prominent work and applications of metaprogramming with C++ templates are reviewed.

Chapter 4 gives an outline of the included publications. In the sequel, we use the numbering adopted in chapter 4 (Roman numerals I–VI) to refer to these publications.

Chapter 5 concludes the thesis and introduces a C++ template library which is largely based on the results of the thesis. Also a brief description of ongoing and future work, which generalises and extends the results of the papers IV–VI, is included.

1.4 The role of C++

Templates were introduced to C++ in order to support conventional generic programming but, quite by accident, they serve as a metaprogramming language as well. As these features were unintentional, the syntax of this sublanguage is awkward. Nevertheless, the existence of such metaprogramming capabilities in a popular programming language, even though in a primitive form, has gained comparatively wide attention (see chapter 3). It is important to explore the capabilities of metaprogramming features in C++ for two reasons: First, they solve real-world problems. Second, metaprogramming features should really be designed properly and added to a programming language intentionally, not by accident. The experience with C++ templates hopefully helps to clarify the requirements for reasonable and useful metaprogramming features in programming languages in general. Naturally this applies to future development of C++ as well.

Chapter 2

Generic programming

Non-generic data structures and algorithms are expressed in terms of unsubstitutable fixed types and constants. *Generic software elements*[1] are parameterised with respect to some of these types and constants. By binding, i.e. *instantiating*, these *formal generic parameters* in different ways, several concrete software elements can be generated from a single generic definition. Hence, generic software elements are schemata which express common behaviour and structure, invariant of the actual values the formal generic parameters are bound to.

2.1 Generic programming in different languages

Several programming languages are equipped with generic programming facilities. CLU [LAB+81] was among the first representatives of such languages, while Ada [TDT97], Eiffel [Mey92, Mey97] and C++ [Str97] are probably the most widely known ones. ML [Pau91] is a well-known functional language with generic features. The support for genericity appears in somewhat varying forms in different languages. Below, the main points of divergence are summarised, briefly describing also the approaches adopted by the five languages mentioned above.

1. The compilation model of generic software elements.

   In C++ and Ada mere generic definitions are not compiled but rather the instantiation results in a more or less textual expansion of the generic code. The outcome is a non-generic compileable version of the code. Eiffel, CLU and ML compilers, on the other hand, compile a common skeleton code from the generic definition of a software element. This implies the necessity of some run-time mechanisms (dynamic binding) for ensuring correct behaviour of the instantiations.

[1] The general term *software element* here refers to any unit of program code, such as a function, subroutine, class or package.

2. Generic parameters.

   In addition to type parameters, some languages support also *non-type* generic parameters, basically compile-time constants. Eiffel has only type parameters, whereas CLU allows constants of any built-in type as generic parameters. Ada supports constant objects of any type, functions and even other generic *packages*[2] as formal generic parameters. C++ supports integral non-type generic parameters, but also references or pointers to objects and functions, pointers to class members and as Ada, other generic classes. The formal generic parameters in ML are *signatures*, which are analogous to class interfaces, and the actual generic arguments are *structures*, implementations of signatures.

3. Generic software elements.

   There is some discrepancy on what software elements can be defined generic. All of the five languages support generic abstract data types (generic classes in Eiffel and C++, packages in Ada, clusters in CLU and functors in ML). Except for Eiffel and ML, the same is true for generic subroutines. C++ and CLU allow even individual member functions (procedures in CLU) of a class (cluster) to be generic. In some respect, ML supports generic subroutines via *polymorphic functions*. Usually no explicit types are given in ML function parameters. ML infers the types from the usage of the parameters. If no type-specific operations are performed with a given function argument the inference system leaves the type of that argument open.

4. Constrains on generic parameters.

   It varies, whether the possible values of generic parameters are restricted explicitly in the generic definition or not. In CLU and Ada, the constrains on the generic parameters are declared in the generic definition. The same approach is taken in Eiffel, where generic parameters are explicitly required to inherit from a given class or classes. ML functors are analogous in this respecet, the generic parameter of a functor must match a predefined signature. Moreover, ML functor definitions may require constrains between different generic parameters. In C++, however, the constrains are implicit. The validity of the values of generic parameters is verified during the compilation of the instantiation.

5. Explicit vs. implicit instantiation.

   Among the five languages, C++ and ML are the only ones supporting implicit instantiation. In ML, a polymorphic function (say an identity

[2]Package is an encapsulated program module in Ada.

function) can be applied to any type. The application does not require the type of the arguments to be specified and thus polymorphic functions are implicitly instantiated.

Similarly in C++, no explicit type declarations are required to instantiate a generic function, rather the generic parameters are deduced from the types of the actual parameters of the function. For example, suppose we have the following generic subprogram declaration in Ada:

```
generic
  type T is private;
procedure Swap(U, V : in out T);
```

Prior to using the subprogram it must be explicitly instantiated and an appropriate actual generic parameter must be provided for the formal generic parameter T. Supposing a and b are variables of type Integer, the generic subprogram can be instantiatiated and used as follows:

```
procedure Exchange is new Swap(Integer);
Exchange(a, b);
```

In C++ the corresponding declaration is

```
template<class T> void swap(T& a, T&b);
```

The instantiation takes place implicitly as a byproduct of a call to the generic function, e.g.:

```
swap(a,b)
```

There are good arguments against any implicit behaviour in programming languages. It may be considered as trading safety for flexibility. However, implicit instantiation is, in essence, a form of polymorphism, which is seldom considered harmful.

6. Specialisation of generic software elements.

   A feature found exclusively in C++ is the possibility to specialise generic classes and functions with respect to the actual values of the generic parameters. In other words, the values of the generic parameters determine which implementation, among a set of alternative implementations, is instantiated (see sections 2.2.2 and 2.2.4).

The primary reason for including generic features to these languages has been to support the implementation of traditional generic containers. If not expressed directly by the language designers [Str94, chapter 15], this becomes clear by the examples demonstrating the use of genericity in the language descriptions [TDT97, LAB+81, Mey97]. As the foremost objective is the same, it can be argued whether the somewhat subtle differences

listed above bear any significance. While comparing the genericity models of C++, Eiffel and the programming language BETA [MMPN93] (mostly with respect to the 4th item above), it has been suggested that in practice all these three languages can express the same, despite the different approaches chosen [Mad95, section 4.3]. The author's perception is, however, that the differences are important and have unexpected consequences. Particularly, the combination of the features selected to C++ result in a genericity mechanism with expressive power surpassing its original intent.

2.2 Generic features in C++

Several language features and design choices contribute to a highly versatile – and complex – genericity model of C++. In addition to ordinary generic containers and algorithms, there is a whole range of possibilities for exploiting genericity. The archetype of this development is the Standard Template Library (STL) [SL94], now part of the C++ Standard Library. For example, various novel generic constructs of STL, such as *functors* and *binders*, provide C++ with higher order functions and a currying mechanism (e.g. [Pau91]) common in functional programming languages, albeit in a restricted form. The use of these constructs with generic STL algorithms can actually be seen as a small shift towards functional programming style. This issue is discussed in paper VI more thoroughly.

Recently, other sophisticated template techniques have emerged, stretching the boundaries of generic programming. It has proven out that templates can, for instance, be used to selectively generate program code, perform non-trivial computations, carry out type mappings and construct data structures at compile time. The remainder of this chapter summarises the template features of C++ that are relevant for understanding these techniques. The most prominent of the techniques are reviewed in chapter 3.

2.2.1 Class templates

Class templates are the parameterised abstract data types of C++. For example, a generic *array* can be defined as follows (showing only the internal structure and omitting all the methods):

```
template<class T, int N>
class array {
  T rep[N];
    ...
};
```

The template keyword is followed by the list of formal template parameters. T is a type parameter, N is a non-type parameter, here of type int. The

formal parameters can be used as such in the template definition: T and N specify the type and number of the elements in the array.

To use the array template, it must be instantiated:

```
class A;
array<A, 10> anArray;
```

The instantiation with the class A and constant 10 generates a class definition array<A, 10> basically by substituting each occurrence of T and N with A and 10 respectively. A generated version of a template is called an *instance* or *instantiation*. The term *specialisation* is also often used. The first terms are used throughout this thesis, the last term being ambiguous in the C++ template vocabulary. The alternative meaning for specialisation adopted here is explained below.

2.2.2 Class template specialisation

Specialisation is a means to provide alternative definitions for class templates to be used with a specific set of arguments. The syntax for defining specialisations is:

```
template<template_parameter_list>
class class_name<specialisation_pattern> { ... };
```

The *template parameter list* declares the template paremeters used in the *specialisation pattern* and is thus not related to the template parameter list of the primary template.

A specialisation is instantiated instead of the general *primary* template, if the values of the template arguments match the specialisation pattern of the specialisation. In an *explicit* specialisation, the specialisation pattern consists of non-generic types and constants. The value of every template parameter is specified exactly, hence the template parameter list is empty. In *partial* specialisation the specialisation pattern defines a *set* of matching values for some template parameters. As an example, consider the following three specialisations for the array template:

```
template<class T, int N> class array { ... }
// The primary template

template<class T> class array<T, 0> { ... }
// #1, partial specialisation for zero-length arrays

template<> class array<bool, 8> { ... }
// #2, explicit specialisation for 8-element bool-arrays

template<class T, int N> class array<T*, N> { ... }
// #3, partial specialisation for pointer type elements
```

To determine which definition is used to generate a particular instantiation, the compiler tries to match the argument list of the instantiation with each specialisation pattern. The specialisations form a partially ordered set, the ordering being defined by the degree of specialisation:

Definition 2.1 *If every template argument list that matches one specialisation also matches another specialisation, but not vice versa, then the first specialisation is said to be* more specialised *than the other, otherwise the two specializations are unordered.*

Based on this partial ordering, a matching specialisation that is more specialised than any other matching specialisation, is instantiated. The detailed rules [C++98, Section 14.5.4] determining the matching and ordering are somewhat complex but nevertheless intuitive. Basically the selection mechanism corresponds to the type *unification* mechanism in compilers [ASU86, chapter 6]. As an example, consider the following instantiations of the preceding array templates:

```
array<A, 10> a;          // primary template
array<bool, 8> b;        // explicit specialisation #2
array<bool*, 8> c;       // partial specialisation #3
array<double*, 0> d;     // error, ambiguous
```

The last instantiation is an error, since there are two matching specialisations (1 and 3), neither of which is more specialised than the other.

2.2.3 Function templates

Analogously to classes, functions can be generic as well. A generic function to compute a minimum of two variables can be defined as follows:

```
template<class T> T min(T a, T b) {
  if (a < b) return a; else return b;
};
```

Note, that the type T is not explicitly restricted in any way, though obviously it must have the '<' operator defined. As explained in section 2.1 (item 4), the constrain is implicit and enforced by the compiler during instantiation. If a template function invocation is encountered in the compilation process, the instantiation occurs as a side effect. The values of the generic parameters can be explicitly specified within the invocation, but if they can be deduced from the types of the function arguments, this is not necessary. The *template argument deduction* succeeds, if the function argument list identifies the set of template arguments uniquely. For example:

```
float pi = 3.14, e = 2.72;
float e_or_pi = min<float>(e, pi);
float pi_or_e = min(pi, e);
```

In the first call to the min function template, the value of the generic parameter T is explicitly specified, whereas in the second invocation the compiler deduces it from the types of pi and e. Both cases generate the same instantiation:

```
float min(float a, float b);
```

2.2.4 Specialising and overloading function templates

Function templates can be given *explicit specialisation definitions* as well as be *overloaded*, just as ordinary functions. The former is roughly equivalent with explicit specialisation of class templates, the latter corresponds to partial specialisation. For example (min_element is a function template in the Standard Library):

```
// explicit specialisation definition for C-style strings
template<> const char* min(const char* a, const char* b) {
  if (strcmp(a,b)<0) return a; else return b;
}
// overloaded definition template<class T>
T min(T* array, int size) {
  return std::min_element(array, array+size);
}
```

The overload resolution is relatively complex and rich in details as overloaded templates, explicitly specialised templates and ordinary functions all take part in the resolution. However, the intuitive workings of the mechanism are clear. The template arguments of each candidate template function are deduced from the argument types of the function call. A partial order among all matching function templates is defined and the most specialised *unique* instance is selected to take part to the overload resolution with ordinary functions. Hence, if two or more overloaded templates match, none of which is more specialised than the others, a compile-time error results. See [LL98, chapter 10] for a thorough description of the overload resolution mechanism.

2.2.5 Compile-time polymorphism

The various specialisation capabilities result in *compile-time polymorphism*, the term introduced in [KS95]. Every template instantiation offers a choice,

which is made during compilation. Based on the values of the generic parameters, the compiler chooses a particular instantiation among a set of alternative definitions, which may be entirely unrelated.

2.2.6 Miscellaneous template features

Apart from compile-time polymorphism, the C++ template mechanism also draws power from several additional features worth mentioning:

- Types and constants (using `typedef`, `enum` and static integral constants) can be defined in class templates.
- Static members, being equivalent to class level variables and functions, are allowed in class templates.
- Template parameters can have default values.
- Member functions of ordinary classes and template classes can be templates.
- Template parameters can be templates.

These features are all important contributors to the versatile, but admittedly also very complex, genericity model of C++.

Chapter 3

Contemporary generic programming in C++

As mentioned in section 2.2, C++ templates extend beyond ordinary generic containers and algorithms. E. Unruh was the first to discover the lurking computational power of templates. He demonstrated it with a program generating prime numbers as a part of compiler's error messages [Unr94].

Rather than being especially planned, the ability to perform computations during compile-time seems to have crawled into the language quite incidentally. Since Unruh's finding, the computational and expressive power of C++ templates has been further explored. Some of the results of this exploration are introduced in this chapter.

3.1 Template metaprogramming

Computations carried out by a compiler can not use dynamic data, rather the inputs, outputs and variables of such computations are constants and types. The template instantiation phase of the C++ compilation process provides a rudimentary mechanism for manipulating these constants and types and controlling program flow. Looping is achieved with recursive template instantiations and selection with template specialisations. With these constructs we can write programs to be executed by the compiler. This type of programming is known as *template metaprogramming* [Vel95b].

3.1.1 C++ basics for template metaprogramming

This section describes the syntax of C++ constructs which are typical in template metaprogramming. A reader familiar with C++ may safely skip the section, comprehensive texts describing C++ include [Str97, LL98].

Integral constants within classes, or instantiated class templates, can be defined either as enumerations with the `enum` construct or as `static` constants.

Types are defined using the typedef keyword. For example, the following template defines integral constants value1, value2, value3 and defines a name (a_type) for the type T:

```
template<class T, int N>
class A {
public:
  enum { value1 = 1, value2 = N };
  static const int value3 = 100;
  typedef T a_type;
};
```

The *scope resolution operator* :: is used to refer to such constants and types. For example:

```
A<int, 10>::value1
A<string, 0>::a_type
```

The first line refers to the constant value1 in the instantiation A<int, 10>, the second to the type a_type in A<string, 0>.

The conditional operator cond ? e1 : e2 is often used in template metaprograms. It corresponds to if (cond) then e1; else e2; with the distinction of being an expression, and thus having a value, rather than being a statement.

Inline expansion is crucial for efficiency in template metaprograms. The inline keyword in front of a function definition instructs the compiler to expand the function body in the call site rather than generate code for calling the function. If a member function is defined directly within the class definition, the inline keyword is not used; the function is inlined by default. Inline expansion is beneficial for small functions, where the performance penalty of the function call and return instructions is significant compared to the function body itself. Template metaprograms typically contain calls to very small (even empty) functions.

For brevity, template metaprograms tend to favor struct definitions for class definitions. Most of the classes in template metaprograms contain only type defintions, constants or static functions and thus do not have a *state* which could change. There is seldom reason to restrict the access to the members of such classes. The access protection of the members of a struct is public by default, opposed to a class where the default protection level is private.

3.1.2 Numerical compile-time computations

The simplest examples of template metaprograms are compile-time computations of numeric constants. For example, the following template definitions comprise a template metaprogram for computing the greatest common divisor of two integer constants.

```
template<int A, int B>
class gcd {
  static const int newA = A<B ? B : A-B;
  static const int newB = A<B ? A : B;
public:
  static const int value = gcd<newA, newB>::value;
};

template<int A> class gcd<A,0> { public: static const int value = A; };
```

The primary template is the general case of the recursive definition, the base case is defined in the partial specialisation for B = 0. The parameters of the metaprogram are the template parameters A and B, the result resides in the value constant. An instantiation, such as const int x = gcd<45,36>::value; instantiates the primary template, computing the static constants newA = 9 and newB = 36. The evaluation of the value constant triggers recursively the instantiation gcd<9,36>::value, which in turn requires gcd<36,9>::value to be instantiated, and so on. At some point, B becomes 0, the specialisation is instantiated and the recursion ends. This all occurs at compile time. The result is the single constant 9 in the executable code.

Numeric template parameters are restricted to integral types; floating point numbers are not allowed. However, even though it may not be directly apparent, it is possible to compute floating point constants at compile time as well. The basic idea is to use rational numbers and series expansions as approximations [GG98, Vel95b].

In principle, the expressive power of the template metaprogramming mechanism allows arbitrary computations (say, a Turing machine simulation) to be performed with a C++ compiler. However, there are of course practical limitations induced by the finite resources available to the compiler.

3.1.3 Metaprogramming with types

Template metaprogramming is not restricted to computations with numbers. Types can be manipulated as well. For example, the following metaprogram implements a compile-time if statement resulting in a selective type definition:

```
template<bool Cond, class Then, class Else>
struct IF { typedef Then RET; };

template<class Then, class Else>
struct IF<false, Then, Else> { typedef Else RET; };
```

Both templates define a type RET which acts as a result of the program. Depending on the value of the boolean constant Cond, either the primary

or the specialised template is instantiated, giving RET either the type Then or Else. As an example of the usage of such a construct, the expression IF< sizeof(A)>sizeof(B), A, B>::RET selects the larger of two types A and B. Using similar techniques, other control structures (case structures, loops) can be defined to guide type and constant selections. The above IF template is an excerpt from [Cza98]. Templates that implement type functions are known as *traits* templates [Mye95].

3.1.4 Compile-time data structures

Compile-time data structures, such as lists and trees, can be represented as nested template instantiations. Such data structures are then manipulated with template metaprograms.

Compile-time lists are in central role in papers II, IV, V and VI. Trees come across, for instance, in *expression templates* that are used to represent parse trees of vector and matrix expressions [Vel95a]. The skeleton of such a template can be written as:

```
template<class Node, class T1, class T2>
struct tree {
  Node node;
  T1 t1; T2 t2;
};

class plus; class times; ... // sample operator types
```

The Node type represents the operator, types T1 and T2 the arguments. As an example, the instantiation tree<plus, tree<times, int, int>, double> could be used to represent the expression type (int*int)+double.

Recursive bind expressions, explained in paper VI, lead to tree structures as well.

3.1.5 Generating code

The result of 'executing' any of the preceding template metaprograms is either a type or a constant. In addition to performing computations, template metaprograms can generate code as well. Code generation is based on selective inlining code along the execution of the metaprogram. The building block code fragments are commonly static member functions, but can be non-member functions or function templates as well. Consider the following template metaprogram:

```
struct check {
  static void execute(int i) { if (i==0) throw Exception; }
};
```

```
struct nocheck {
  static void execute(int i) { }
};

IF<check_flag, check, nocheck>::RET::execute(i);
```

check and nocheck are structs, which both contain a member function execute with different definitions. IF<check_flag, check, nocheck>::RET expands either to the type check or nocheck depending on the value of check_flag (see section 3.1.3). Consequently, the value of check_flag determines which of the execute functions the last line invokes. Note that in the case where check_flag == false the empty function does not yield any code.

To perform code generation in a larger scale, there must be some kind of iteration involved. The following example, taken from [Vel99] generates a specialised dot product algorithm:

```
template<int I>
inline float dot(float a[], float b[]) {
  return dot<I-1>(a,b) + a[I]*b[I];
};

template<>
inline float dot<0>(float a[], float b[]) {
  return a[0]*b[0];
};

// Example:
float x[3], y[3];
float z = dot<2>::f(x,y);
```

The last line generates code that is equivalent to:

```
float z = x[0]*y[0] + x[1]*y[1] + x[2]*y[2];
```

Hence, the template metaprogram was used to achieve *loop unrolling*.

Iteration can of course be over types in trees and lists etc. The articles I, II, IV, V and VI show several examples of such code generation.

3.2 Applications of template metaprogramming

The publications in this thesis describe generative programming techniques and active libraries. The underlying implementation technology is template metaprogramming. This section reviews some of the related work describing successful active libraries using template metaproramming.

3.2.1 Object synthesis and configuration repositories

Czarnecki and Eisenecker [CE99b] have proposed a method for synthesising objects from *implementation components*. These components implement certain functionalities and can be configured to yield different systems. The functionalities can be aspects (see section 1.1.2), i.e., such that they can not be encapsulated into a single procedure or object. An example from [CE99b] presents a configurable list container consisting of implementation components for basic list functionality, logging insertions and deletions, element counting, element copying, element ownership etc. These implementation components can be composed and configured to yield concrete list implementations with desired characteristics.

Without going into further details, template metaprograms map an abstract configuration description to a concrete *configuration repository*, which guides the object synthesis. For example, the abstract specification

```
LIST_GENERATOR<A, copy, with_counter, with_logging>::RET
```

defines a list type that contains elements of type A, stores them as copies (instead of references), keeps count of the number of elements and logs insertions and deletions.

3.2.2 Generic programming in linear algebra

Three articles in this thesis (papers I, II and III) deal with sparse matrix computations. To get a broader insight into the area, three other matrix computation libraries that utilise template metaprogramming are discussed in this section.

Linear algebra is a fruitful application area for the new generic programming techniques. The domain specific language of matrix expressions is well-understood and the need for high performance is characteristic for these libraries. There is a loose community with an interest on linear algebra and scientific computing using OOP and generic programming (see *The Object Oriented Numerics* home page [OON00]).

Traditional high performance linear algebra and matrix computation libraries, such as LAPACK [ABB+99] are typically written in Fortran or C. Such libraries are very large; there are different versions of the same routines for several precision types (single and double precision real, single and double precision comblex), few dense storage types (general, banded, packed), and a large number of sparse storage types [SL98]. Covering all these cases is a combinatorial issue. Furthermore, to achieve high performance several special algorithms are provided for commonly encountered simple expressions. For example, instead writing Y = A*X + Y to multiply the vector X with the constant A and add another vector Y to the result, a FORTRAN programmer would call the BLAS (basic linear algebra subroutines)

library [LHKK79, DDHH88, DDDH90] routine SAXPY, DAXPY, CAXPY or ZAXPY (depending on the precision) as SAXPY(N, A, X, 1, Y, 1). To complicate things more, the processor characteristics and the memory architecture have a significant effect on performance. This is why processor manufacturers commonly supply specially tuned versions of the BLAS libraries.

There are a number of numerical libraries for OO languages (C++ [Poz99, Mat99], Java [JAM99, JNL]). These libraries provide some improvement over the C and FORTRAN libraries, e.g. by parameterising the precision of elements, but they still more or less retain the structure of the original libraries: each special case (different matrix formats etc.) requires a special algorithm.

Straightforward attempts to use OO principles to abstract matrix computations lead to very inefficient solutions. For example, creating an abstract superclass for all types of matrices and binding the element access function dynamically is not acceptable on most cases. Another source of inefficiency are the temporary variables created during the pairwise evaluation of expressions. In a straightforward implementation with operator overloading, an expression like a = b + c + d (where a, b, c and d are vectors or matrices) sets off the construction of two unnecessary temporary variables for holding the results of the subexpressions b + c and b + c + d.

The semantic gap between the mathematical abstract matrix notation and the efficient implementation is wide. The knowledge that is needed to transform the mathemathical expressions to an effient implementation is considerable. To aid these tasks, several active libraries have been developed demonstrating the strength of generic, and generative, programming in this field.

3.2.3 MTL — The Matrix Template Library

MTL [SL98] is a high-performance generic linear algebra library written in C++. It borrows its structure form STL, consisting of generic functions, containers, interators, adaptors, and function objects. MTL does not currently provide operator overloading abstractions. It implements the functionality of BLAS libraries plus some more.

The main focus of the library is in covering different precisions, different storage types and different memory layouts with the same configurable algorithms and containers. This is achieved with the configuration repository mechanism discussed in section 3.2.1. The element precision, storage types etc. are configuration parameters for the matrix type. E.g. matrix<double, rectangle<>, dense<>, column_major>::type defines a double precision dense rectangular matrix with column major storage order. This configuration information can be used while instantiating the generic matrix algorithms to provide specialised algorithms for certain types of matrices. However, for the most part, each algorithm is implemented with just

one generic algorithm, which takes advantage of the configuration information to generate an efficient implementation. This results in very high degree of code reuse. The size of the MTL library is only 15 KLOC[1] compared to the 150 KLOCs of the FORTRAN BLAS while providing greater functionality and better performance.

Another interesting feature in MTL is the use of template metaprograms to tune itself for different memory hierarchies. To obtain high-performance, elementary matrix algorithms must take advantage of the memory hierarchy of the underlying processor. This is achieved by careful loop blocking and structuring (e.g. [DS98]). Optimal block sizes depend on the number of registers and cache sizes. Based on these few simple constants MTL can generate matrix-matrix multiplication code with performance on par with the vendor-tuned routines.

3.2.4 The Generative Matrix Computation Library

The Generative Matrix Computation Library (GMCL) [Cza98] is somewhat similar in structure to the MTL; the configuration scheme of matrices is analogous. The GMCL clearly separates the domain specific configuration language and the implementation language: however, both are written as C++ templates and executed by the compiler.

The concrete matrix types are specified using the configuration language. The set of configurable features is even more complete than in the MTL including for example error bounds checking and various memory allocation schemes. GMCL overloads arithmetic operations for matrices efficiently using *expression templates* (see section 3.2.5).

The GMCL implementation comprises of 7500 lines of code but is able to cover more than 1800 different matrix types. The performance of the generated code is comparable with the performance of the manually coded variants.

3.2.5 Expression templates and Blitz++

The results of the pioneering work by Veldhuizen [Vel95b, Vel95a] with template metaprogramming and expression templates is the basis of the Blitz++ library [Bli99, Vel98]. The library solves the problem of unnecessary temporary creation in vector and matrix expressions with the expression templates technique. Template metaprogramming is exploited to generate specialised algorithms for small vectors and matrices. In addition to these, the library performs several domain and architecture specific optimisations.

Expression templates break the normal pairwise evaluation of the arithmetic expressions. Arithmetic operators are overloaded for vectors and matrices to build parse trees. For example, the result of the expression `a + b`,

[1] Thousand lines of code

where a and b are vectors, is not a vector but rather an expression object (see section 3.1.4) holding pointers to a and b. Hence, the evaluation of an arbitrary matrix or vector expression creates a parse tree of the expression. When such a parse tree is assigned to some vector or matrix, the actual computation is performed without creating any temporaries. This is possible, since the assignment operation can use metafunctions to analyse the parse tree and generate efficient code for evaluating the expression of such a type. For recent work on expression templates, see [Fur97, HCKS99].

3.2.6 Miscellaneous applications

Template metaprograms have been used to provide various small-scale abstractions. For example, computing the required memory space for an n-dimensional array, while n is a generic parameter, can be done with a template metaprogram [Vel99]. Another example is the selection of the most appropriate integral type for representing a given number of bits. The C++ standard does not fix (entirely) the sizes of integral types, such as char, int, long etc. Hence, the most efficient type may vary from platform to platform. A template metaprogram can perform this selection in a portable way [Pes97].

As a demonstration of the expressive power of template metaprogramming, a rudimentary LISP implementation has been written with C++ templates [CE99a]. Common compile-time control structures (if, while, switch etc.) have been defined to make template metaprogramming easier [CE99a]. The compile-time if-structure was described in section 3.1.3.

3.3 Restrictions of template metaprogramming

Template metaprogramming has gained popularity as an implementation mechanism for generative programming. An important reason for this is the wide availability of the mechanism: a standard C++ compiler is enough for interpreting template metaprograms. However, template metaprogramming suffers from many deficiencies:

- There is no support for debugging.
- The diagnostic systems of compilers do not usually cope well with extensive usage of templates [Ale99]. Particularly the error messages from template metaprograms tend to be lengthy and difficult to interpret.
- Compilation times increase (see papers IV and V).
- Though template metaprogramming strives for highly intentional abstractions, template metaprogramming itself is far from intentional.

The syntax is sometimes unwieldy, metaprograms are formed by non-encapsulated fragmented pieces of template definitions, interactions with the core language are complex etc. The compile-time control structures referred to in section 3.2.6 are of some help in this respect.

- At the time of writing this, some widely used compilers are still far from standard conforming. The non-conformant features are typically related to templates and thus template metaprogramming.
- There are some language details, which restrict the mechanism. For example, the *size* of an arbitrary expression can be queried with the sizeof operator, but there is no means to query the *type* of an expression.

Despite the obvious shortcomings, a number of successful applications using template metaprogramming have been built and there is a growing interest towards the technique. This is an indication of the importance and usefulness of metaprogramming facilities in programming languages in general.

Chapter 4

Summary of publications

The thesis constitutes of six publications. They present template metaprogramming techniques and their applications in a rather technical and detailed level. The first three communications provide efficient abstractions for numerical domain. The latter three papers describe active libraries which define new constructs for the C++ language itself. This chapter outlines the contents of the included articles. The publications are:

I. [Jär97] Jaakko Järvi. Processing Sparse Vectors During Compile Time in C++. In *Scientific Computing in Object-Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, pages 41–48. Springer-Verlag, 1997.

II. [Jär98] Jaakko Järvi. Compile Time Recursive Objects in C++. In *Technology of Object-Oriented Languages and Systems*, pages 66–77. IEEE Computer Society Press, 1998.

III. [Jär99a] Jaakko Järvi. Object-Oriented Model for Partially Separable Functions in Parameter Estimation. *Acta Cybernetica*, 14(2):285-302, 1999.

IV. [Jär99b] Jaakko Järvi. Tuples and Multiple Return Values in C++. Technical Report 249, Turku Centre for Computer Science, March 1999.

V. [Jär99c] Jaakko Järvi. ML-style Tuple Assignment in Standard C++ — Extending the Multiple Return Value Formalism. Technical Report 267, Turku Centre for Computer Science, March 1999.

VI. [Jär99d] Jaakko Järvi. C++ Function Object Binders Made Easy. In *Proceedings of the Generative and Component Based Software Engineering 99*, September 1999. To appear in volume 1799 of Lecture Notes in Computer Science.

The combined results of the papers IV and V have been presented in the *Nordic Workshop on Programming Theory'99* and published as an extended abstract in the workshop proceedings [Jär99e]. Further, a combined and revised version of the two papers is to appear in the *C++ Report* journal.

I Processing Sparse Vectors during Compile Time in C++

The paper describes a novel programming technique for performing sparse vector index computations at compile time. The technique uses the bits of integral template parameters to express the sparseness structure of vectors in the type of the vector objects, i.e., vectors with different sparseness structures are of different types.

The technique is best suited for small vectors where the sparseness structure is known at compile time. The paper outlines an active library that, under the above conditions, allows the programmer to write programs which use abstract vector expressions and still reach equal performance to hand optimised low-level C-code. The user gains a higher intentionality level without sacrificing performance.

Template metaprograms define the mapping from the vector expressions to an efficient implementation. More concretely, template metaprograms unroll the loops over vectors and eliminate multiplications with zero. The paper describes *forward mode automatic differentiation* (AD) as an application of the library. Automatic differentiation relieves the programmer from writing explicit derivative code for mathematical expressions but still retains better precision compared to approximated difference values. The use of the proposed vector presentation in AD computations gave superior performance compared to more conventional vector representations. Furthermore, the execution speed was very close to that of the corresponding symbolically differentiated expressions.

II Compile Time Recursive Objects in C++

The article is a generalisation of the paper I in several ways. The sparse vector templates in paper I are basically lists, where a certain bit of a template parameter determines the type of a list element. Hence, for each element there is a binary choice between two predefined types. This paper describes a more general list structure, where each element type can be chosen freely from the set of all possible types. Compile-time indexing of sparse vectors, as presented in paper I, is thus a special case of such compile-time lists. Regarding this application, the data structures generalise directly to matrices and higher dimensional arrays as well.

The paper describes how to define compile-time lists using *recursive templates*, that is, templates which refer to other instantiations of the same template. The paper shows how to manipulate such data structures and how to use them in code generation. The presented results lay an important foundation for the later articles IV-VI.

III Object-Oriented Model for Partially Separable Functions in Parameter Estimation

The perspective of the paper differs somewhat from that of the five other papers in the thesis; generic or generative programming is not particularly accentuated. However, generic programming plays an important role in the core of the computational kernel described in the paper. The kernel uses automatic differentiation in derivative computations. In this task the template metaprogramming approach to sparse vector indexing described in paper I is used. The efficient indexing scheme had a significant effect on the performance results reported in the article.

Parameter estimation is an optimisation process where a model function depending on modifiable parameters is fitted to a set of data points. To assess the goodness of the fit, we must be able to compute the values of the model function. Further, for the optimisation process to be efficient, we commonly need a means to compute the derivatives with respect to the modifiable parameters. The paper describes an object oriented framework for representing partially separable model functions. Such functions are commonly encountered, e.g. in spectroscopy. The framework provides an efficient computational kernel for computing the model function values and derivatives. The model functions are represented in a structured and intuitive way, resembling the mathematical structure of the functions. The formulae of the model function are clearly encapsulated into few very simple classes. Furthermore, the framework utilises automatic differentiation in derivative computations. We thus attain a highly intentional, yet efficient, representation for model functions.

The paper includes a case study of *nuclear magnetic resonance* (NMR) spectral fitting. See [JNKF97] for details of this problem domain, as well as a description of an NMR analysis program based on the presented framework.

IV/V Tuple types in C++

A tuple (or n-tuple) type is the cartesian product of its element types. In a programming language, a tuple is a data object containing a fixed number of other objects as elements. These element objects may be of different types. Tuples are convenient in many circumstances, in particular they make it easy to define functions that return more than one value.

Unlike some programming languages, such as Python [Pyt99, Lut96], ML [Pau91] and Haskell [Tho99], C++ does not provide the user with built-in tuple types. The fourth and fifth paper propose a tuple construct for C++. This construct is a generalisation of the C++ Standard Library pair template from two to arbitrary number of elements.

The tuple abstraction is based on compile-time lists. Template metaprograms aid at constructing tuples and accessing tuple elements. The two papers IV and V together outline an active library which implements a domain specific abstraction — here the domain is the programming language itself.

The papers compare different ways of passing multiple results out of a function and stress the conciseness and clarity of tuple typed return values compared to extra output parameters of pointer or reference types. As a concrete example, let foo be a function with two inputs (of some types A and B) and three outputs (of some types C, D and E). A typical prototype and a corresponding call for such a function could be:

```
void foo(A, B, C&, D&, E&)
        ...
foo(a, b, c, d, e);
```

Here a, b, c, d and e are of types A, B, C, D and E respectively. The proposed tuple abstraction allows a more intentional definition and use:

```
tuple<C, D, E> foo(A, B);
        ...
tie(c, d, e) = foo(a, b);
```

The papers discuss the performance of different design alternatives at compile and run time and show that with an optimising C++ compiler the extra runtime cost arising from the abstraction is negligible, and that the extra compile time cost is notable, but not significant on real programs.

IV/V.1 Compilation tests

Since writing the papers IV and V, advances in compiler technology have been introduced. Specifically, the inlining analysis in the GCC C++ compiler is performed at an earlier phase of the compiling process. As a result, compilation times decrease in programs that make heavy use of inline functions.

This is an important improvement, since programs using STL containers and algorithms fall in this category. A frequent use of inline functions is even more typical for the template techniques described in this thesis. To assess the effect of the new inliner the compilation tests of the paper V were rerun with a new version of the GCC C++ compiler (development snapshot

20000426 of gcc version 2.96). The version in paper V was egcs 1.1.1. Despite the different names, gcc and egcs are different versions of the same compiler.

The goal of the compilation tests was to measure the relative cost of using tuples as function return types compared to using reference parameters to pass values out of functions. Hence, five pairs of programs were generated for different tuple lengths. Each pair contained a program using tuples and a program using reference parameters as the return mechanism. More concretely, the two-elemement case consisted of repeated function definitions of the following form:

```
// Reference parameters
struct A {};
void f(A& a0, A& a1) { a0=A(); a1=A(); }
A g() { A a0; A a1; f(a0,a1); return a1; }

// Tuples
struct A {};
tuple<A,A> f() { return make_tuple(A(),A()); }
A g() { A a0; A a1; tie(a0, a1) = f(); return a1; };
```

The rationale behind these definitions is as follows. In both cases, the function f has many return values (two in this case) and function g calls f, a function with many return values. Hence, we cover both the definition and invocation. The functions f and g should be as simple as possible, yet forcing the compiler to really compile the functions rather than avoid a significant deal of the compilation by detecting that some code is not used at all.

The test programs and settings were identical to the ones described in section 4.1.2 of paper V (optimisation flag -O2, Intel 133 Mhz Pentium). The test results are illustrated in Fig. 4.1. To eliminate the effect of any constant costs the compilation time of a program containing an empty main function was subtracted from the actual compilation times to get T_{ref} values. The compilation time of a program containing an empty main function and including the tuple header file was substracted to get T_{tier} values (on the target machine, the compilation of the tuple header took 0.41 seconds). The standard deviations in repeated tests were negligible (below 1 %) and are not shown.

The tests show a significant reduction in the cost of compiling tuple constructs. This is in accordance with the experiences reported at GCC homepages [GCC00, follow the link: *News, Dec 05*] and thus indicates that the reduced compilation costs are not limited to specific constructs but are of benefit to a wide range of generic programming techniques.

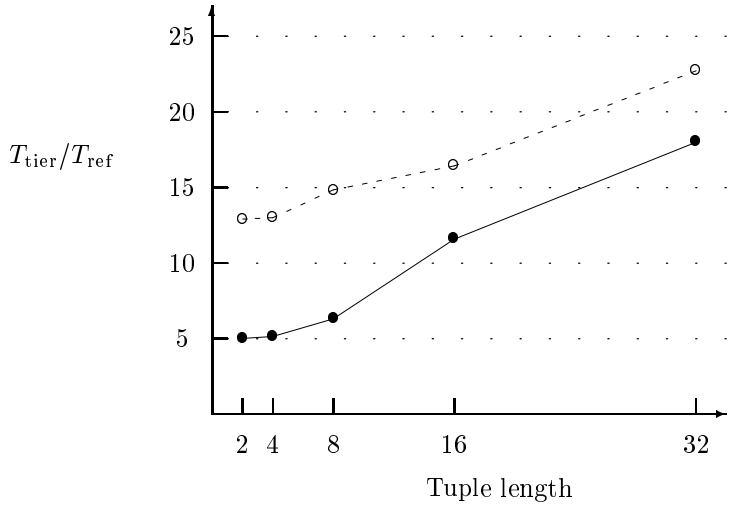


Figure 4.1: The relative compilation times of programs using tuples to equivalent programs using reference parameters to pass multiple values out of functions. T_{tier} is the compilation time of the tuple program, T_{ref} the compilation time of the reference parameter program. The solid line represents the results with the gcc 2.96 20000426 compiler, the dashed line the results with the egcs 1.1.1. compiler.

VI C++ Function Object Binders Made Easy

The last paper describes an active library which extends C++ with another new construct. The C++ Standard Library provides a very rudimentary partial function application (*argument binding* in C++ vocabulary) mechanism with several unnecessary constrains and restrictions. This paper describes a novel argument binding mechanism resembling the partial function application semantics of Theta [LCD+95]. In this mechanismm, the actual arguments to a function can contain explicit *free* pseudo-variables, called *placeholder objects*, allowing thus arbitrary arguments to be bound or left unbound. For example, assume again that a, b, c, d and e are variables of some types A, B, C, D and E, and that E foo(A, B, C, D) is some function. Now foo can be invoked partially as follows:

```
bind(foo, a, free1, c, free2);
```

The first and third parameter are bound to a and c, respectively. The second and fourth parameters are left unbound (free1 and free2 are predefined placeholders). The result of the expression is a function of type E foo2(B, D). When invoked with, say b and d, this function eventually calls the original function as foo(a, b, c, d).

The proposed new binding technique removes several constrains and special cases from argument binding. The bind expressions retain the resemblance to a normal non-bound call of the underlying function, unlike in the binding mechanism in the Standard Library. Further, the new binding syntax and semantics is more expressive allowing function composition.

The paper presents the technique from the users point of view and describes the general layout of the library that implements it.

Chapter 5

Conclusion

5.1 Summary

High intentionality of a programming language is of utmost importance while pursuing for maintainable, reusable and adaptable software. We argued that the currently established abstraction mechanisms in general purpose programming languages tend to fall short in this respect. In an ideal case programming could be done with a language suited best for a particular domain. We stated the problems of providing such domain specific languages and compilers and identified generative programming as a more tractable alternative. Compilation process is opened up to some extent to allow the programmer to define new language constructs, optimisations etc. to attain higher intentionality.

From different implementation technologies for generative programming we focused on generic programming and particularly on template metaprogramming in C++. The basics of template metaprogramming were described and the most prominent applications were reviewed. The individual articles constituting the main body of the thesis explored the template metaprogramming mechanism and demonstrated that generic programming is more than just parameterising the element types in a container library. The articles presented new results in the field of linear algebra and object oriented numerics, as well as provided extensions to the C++ language itself.

Template metaprogramming is obviously not a final solution while attempting to attain high intentionality. This becomes clear from the immediate problems that we identified (awkward syntax, lack of debugging tools, unwieldy error messages etc.). Nevertheless, template metaprogramming solves practical problems today and is readily usable as a byproduct of a standard-conforming C++ compiler. Further, the success and relative popularity of a somewhat awkward, limited and complex metaprogramming facility, such as C++ templates, is an encouragement for equipping programming languages with well-thought generative abstraction tools. The goal of

such a facility is obviously not to make all programmers define their own language prior to writing an application program but rather to bring the degree of difficulty of writing domain specific abstractions from the level of compiler development closer to that of writing a class library.

5.2 The Lambda Library

The results of the last three papers IV-VI have been combined to a template library, together with some new results. This library, called *The Lambda Library* (LL), is available for users at the LL home page [LL00]. The source code and documentation are downloadable from the home page.

The library is written entirely in standard C++. The library contains some improvements, particularly in the tuple implementation, which are not described in the papers included in this thesis. For instance, the library implementation unifies the `cons` template of paper IV and the `ref_cons` template of paper V resulting in a single tuple construct, which is capable of storing elements of practically any valid C++ type. The parts of the functionality of LL not covered in this thesis are described in the documentation accessible from the library home page.

5.3 Future Work

The work with the Lambda Library continues. It builds on the ideas from this thesis and on the work of G. Powell and P. Higley [PH00]. At the time of writing this, we (joint work with Powell) have generalised the partial function application mechanism from paper VI and extended it to cover operator invocations. In effect, this adds unnamed functions, or *lambda functions*, to C++. A description of the functionality and implementation of this part of the LL is planned to be published later. Here we summarise the main features of the lambda abstraction part of the library.

Free variables can be used not only in functions but in arbitrary expressions as well. As an example, the expression `free1 * 10` evaluates to a unary function. When invoked, this function returns its argument multiplied with `10`. Hence, `(free1 * 10)(5)` evaluates to `50`.

The library supports almost all overloadable operators of C++. For example, to compute the dot product of two containers `a` and `b`, one could write:

```
double sum = 0;
for_each(a.begin(), a.end(), b.begin(), sum += free1 * free2);
```

In addition to operators, common control structures can be emulated. Consequently, this provides a means to pass fragments of code as parameters to functions. The following code prints those elements of `a` that are *even*

and *smaller than 100* (&& is the logical *and* operator, % is the remainder operator):

```
for_each(a.begin(), a.end(),
        if_then(free1 < 100 && (free1 % 2) == 0,
                cout << free1 << endl));
```

Furhermore, constructor and destructor calls can be written as lambda functions, as well as memory allocation and deallocation requests. Lambda functions can even throw exceptions and contain try and catch blocks for exception handling.

Lambda expressions can be combined arbitrarily with the bind expressions described in paper VI. Furthermore, the bind expressions themselves have been generalised from paper VI. As stated above, a function such as E foo(A, B, C, D) can be invoked partially:

```
bind(foo, a, free1, c, free2);
```

The current LL allows the target function to be left open as well:

```
bind(free1, a, b, c, d);
```

The result of this expression is a unary function, which takes a four-argument function as its parameter. Hence, the expression

```
bind(free1, a, b, c, d)(foo)
```

invokes foo(a, b, c, d). A bound or unbound target function can be a pointer or reference to a function, a function object (a class object with a function call operator member) or a pointer to a member function.

The current LL provides an alternative for the Standard C++ Libary binders. It is fair to say, that the LL binding mechanism is more intentional and less constrained of these two. There is no performance difference between the two approaches.

With the current Standard Library, particularly with the function objects and binders, C++ has taken a firm step towards an entirely new programming style. The LL is another step in the same direction. The C++ standard is currently frozen but will be reopened for changes after some years. The author believes that some of the features of the LL would be worthy candidates for additions to future revisions of the C++ Standard Library.

References

[ABB+99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 3rd edition, 1999.

[AG96] K. Arnold and J. Gosling. *The Java^{TM} Programming Language*. Addison-Wesley, Reading, MA, 1996.

[Ale99] A. Alexandrescu. Better template error messages. *C/C++ Users Journal*, March 1999.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[Big97] T. J. Biggerstaff. A perspective of generative reuse. Technical Report MSR-TR-97-26, Microsoft Reserarch, December 1997.

[Bli99] The Blitz++ library home page. http://www.oonumerics.org/blitz/, 1999.

[Bos97] J. Bosch. Delegating compiler objects: Modularity and reusability in language engineering. *Nordic Journal of Computing*, 4(1):66–92, 1997.

[Bos98] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, May 1998.

[C++98] International Standard, Programming Languages – C++, ISO/IEC:14882, 1998.

[CE99a] K. Czarnecki and U. Eisenecker. Meta-control structures for template metaprogramming, 1999. http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm.

[CE99b] K. Czarnecki and U. Eisenecker. Synthesizing objects. In R. Guerraoui, editor, *ECOOP'99 – Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 18–42, June 1999.

[CEG+98] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. Generative programming and active libraries. Lecture Notes in Computer Science, April 1998. To appear, also in http://extreme.indiana.edu/~tveldhui/.

[CES97] K. Czarnecki, U. Eisenecker, and P. Steyaert. Beyond objects: Generative programming, a position paper. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97*, June 1997. http://wwwtrese.cs.utwente.nl/aop-ecoop97.

[Chi95] S. Chiba. A metaobject protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, pages 285–299, October 1995.
http://www.hlla.is.tsukuba.ac.jp/~chiba/openc++.html.

[Cox90] B. J. Cox. There *is* a silver bullet. *Byte*, pages 209–218, October 1990.

[Cza98] K. Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* PhD thesis, Technische Universität Ilmenau, Germany, 1998.

[DDDH90] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[DDHH88] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):18–32, 1988.

[DNW+99] S. Dobson, P. Nixon, V. Wade, S. Terzis, and J. Fuller. Vanilla: an open language architecture. In *Generative and component-based software engineering*, volume 1799 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999. To appear.

[DS98] K. Dowd and C. Severance. *High Performance Computing.* O'Reilly & Associates, 2nd edition, 1998.

[Eis97] U. Eisenecker. Generative programming (GP) with C++. In H. Mössenböck, editor, *Modular Programming Languages, Proceedings of JMLC'97*, pages 351–365, Linz, Austria, March 1997. Springer-Verlag.

[Fur97] G. Furnish. Disambiguated glommable expression templates. *Computers in Physics*, 11(3):263–269, May/June 1997.

[GCC00] The GCC home page. http://www.gnu.org/software/gcc, 2000.

[GG98] J. Gil and Z. Gutterman. Compile time symbolic derivation with C++ templates. In *COOTS'98, 4th USENIX Conference on Object Oriented Technologies and Systems*, April 1998.

[GHJ+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[Han94] S. W. Haney. Is C++ fast enough for scientific computing? *Computers in Physics*, 8(6):690–694, November 1994.

[HCKS99] S. W. Haney, J. Crotinger, S. Karmesin, and S. Smith. Pete, the portable expression template engine. *Dr. Dobb's Journal*, pages 88–95, October 1999. http://www.acl.lanl.gov/pete.

[IP99] Intentional Programming home page. http://www.research.microsoft.com/research/ip, 1999.

[JAM99] Mathworks, Inc. and the National Institute of Standards and Technology (NIST). *JAMA: A Java Matrix Package*, 1999. http://math.nist.gov/javanumerics/jama/.

[JNKF97] J. Järvi, S. Nyman, M. Komu, and J. J. Forsström. A PC-program for automatic analysis of NMR spectrum series. *Computer Methods and Programs in Biomedicine*, 52:213–222, 1997.

[JNL] *JNL[tm] 1.0 - A numerical Library for Java[tm]*.

[Jär97] J. Järvi. Processing sparse vectors during compile time in C++. In *Scientific Computing in Object-Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, pages 41–48. Springer-Verlag, 1997.

[Jär98] J. Järvi. Compile time recursive objects in C++. In *Technology of Object-Oriented Languages and Systems*, pages 66–77. IEEE Computer Society Press, 1998.

[Jär99a] J. Järvi. Object-oriented model for partially separable functions in parameter estimation. *Acta Cybernetica*, 14(2):285–302, 1999.

[Jär99b] J. Järvi. Tuples and multiple return values in C++. Technical Report 249, Turku Centre for Computer Science, March 1999.

[Jär99c] J. Järvi. ML-style tuple assignment in standard C++ — extending the multiple return value formalism. Technical Report 267, Turku Centre for Computer Science, March 1999.

[Jär99d] J. Järvi. C++ function object binders made easy. In *Proceedings of the Generative and Component-Based Software Engineering'99*, volume 1799 of *Lecture Notes in Computer Science*, September 1999. To appear.

[Jär99e] J. Järvi. Incorporating tuple types to C++. In *Proceedings of the Nordic Workshop on Programming Theory'99*, October 1999. Extended abstract.

[KL93] D. E. Knuth and S. Levy. *The CWEB System of Structured Documentation*. Addison-Wesley, Reading, MA, 1993. Also at www-cs-staff.Stanford.EDU/~knuth/cweb.html.

[KLM+94] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP'97 – Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, June 1994.

[Knu92] D. E. Knuth. *Literate Programming*. Number 27 in Center for the Study of Language and Information Lecture Notes. CLSI Publications, Stanford University, 1992.

[KS95] A. Koenig and B. Stroustrup. Foundations for native C++ styles. *Software – Practice and Experience*, 25(S4):S4/45–S4/86, December 1995.

[LAB+81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*. Number 114 in Lecture Notes in Computer Science. Springer-Verlag, NY, 1981.

[LCD+95] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A. C. Myers. Theta reference manual, preliminary version. Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, 1995. http://www.pmg.lcs.mit.edu/Theta.html.

[LHKK79] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Transactions on Mathemathical Software*, 5(3):308–323, 1979.

[LL98] S. B. Lippman and J. Lajoie. *C++ Primer*. Addison-Wesley, Reading, MA, 3rd edition, 1998.

[LL00] The Lambda Library home page. http://lambda.cs.utu.fi, 2000.

[Lut96] M. Lutz. *Programming Python*. O'Reilly & Associates, 1996.

[Mad95] O. L. Madsen. Open issues in object-oriented programming - a Scandinavian perspective. *Software - Practice and Experience*, 25(S4):S4/3–S4/43, December 1995.

[Mat99] Math.h++: Object-oriented library for numerical computation, 1999. Rogue Wave Software Inc., Boulder, CO, http://www.roguewave.com.

[Mey92] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[Mey95] B. Meyer. Reusable software: The base object-oriented component libraries (ISE Eiffel, The Libraries). Technical Report TR-EI-44/L, Interactive Software Institute (ISE), 1995.

[Mey97] B. Meyer. *Object Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[MKL97] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009 P9710044, Xerox PARC, February 1997. http://www.parc.xerox.com/csl/groups/sda.

[MMPN93] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.

[MN99] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999. http://www.mpi-sb.mpg.de/~mehlhorn.

[Mye95] N. C. Myers. A new and useful template technique: 'traits'. *C++ Report*, 7(5):32–35, 1995.

[OON00] The object-oriented numerics page. http://www.oonumerics.org, 2000.

[Pau91] L. C. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

[Pes97] C. Pescio. Template metaprogramming, make parameterized integers portable with this novel technique. *C++ Report*, 9(7), July/August 1997.

[PH00] G. Powell and P. Higley. Expression templates as a replacement for simple functors. *C++ Report*, 2000. To appear.

[Poz99] R. Pozo. Template Numerical Toolkit - a numeric library for scientific computing in C++, 1999. http://math.nist.gov/tnt/.

[Pyt99] The Python homepage. http://www.python.org, 1999.

[Sim95] C. Simonyi. The death of computer languages, the birth of intentional programming. Technical Report MSR-TR-95-52, Mircrosoft Research, September 1995.

[SL94] A. A. Stepanov and M. Lee. The standard template library. Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories, April 1994. (http://www.hpl.hp.com/techreports).

[SL98] J. G. Siek and A. Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In D. Caromel, R. R. Oldehoeft, and M. Tholburn, editors, *Computing In Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, 1998. http://www.lsc.nd.edu/research/mtl.

[Str94] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA, 1994.

[Str97] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 3rd edition, 1997.

[TDT97] S. Tucker Taft, Robert A. Duff, and T. Taft, editors. *Ada95 Reference Manual : Language and Standard Libraries : International Standard ISO/IEC 8652;1995(E)*. Number 1246 in Lecture Notes in Computer Science. Springer, November 1997.

[Tho99] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, Reading, MA, 2nd edition, 1999.

[Unr94] E. Unruh. Prime number computation. Distributed in the ANSI X3J16-94-0075/ISO WG21-426 meeting, 1994.

[Vel95a] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[Vel95b] T. L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[Vel98] T. L. Veldhuizen. Arrays in Blitz++. In D. Caromel, R. R. Oldehoeft, and M. Tholburn, editors, *Computing In Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, 1998.

[Vel99] T. L. Veldhuizen. C++ templates as partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, 1999.

[Web95] B. F. Webster. *Pitfalls of Object-Oriented Development.* M & T Books, NY, 1995.

Publication reprints

Processing Sparse Vectors During Compile Time in C++

Jaakko Järvi

Turku Centre for Computer Science, Finland

Abstract. A C++ template library for a special class of sparse vectors is outlined. The sparseness structure of these vectors can be arbitrary but must be known at compile time. In this case it suffices to store only the nonzero elements of the vectors, and no indexing information about the sparseness pattern is required. This information is contained in the type of the vector as a non-type template parameter. It is shown how common vector operators can be overloaded for these vectors. When compiled the operators yield code which performs only the necessary elementary operations between the nonzero elements with no run-time penalty for indexing. All indexing is performed at compile time, resulting in very fast execution speed. The vector classes are best suited for short vectors up to few dozens of elements.
Automatic differentiation of expressions is given as an example application. It is shown how classes for automatically differentiable numbers can be defined with the library. A comparison against other vector representations gave superior results in execution speed of differentiating a few common expressions, and came very close to the calculation speed of symbolically differentiated expressions.

1 Introduction

Templates are a powerful feature of C++. They are usually used to write generic classes and functions, but we can go beyond that. It is possible to make the compiler perform as an interpreter at compile time. For instance, compile-time bounded loops and branching statements can be written using recursive template definitions and template specialisation. These templates are called *template metaprograms* [1]. The use of non-type template parameters is a key factor behind template metaprograms.

In this article template metaprograms are used in the definition of *compile-time sparse vector* (CTSV) classes. The term *compile-time* here means that the sparseness pattern of the vectors (the positions of zero and non-zero elements) is known at compile time. Due to this restriction CTSVs do not serve as general purpose sparse vectors, but they are very efficient in special applications where the above requirement can be satisfied.

The definitions for CTSV template classes are given. It is shown how common operations can be overloaded for CTSV types. Automatic differentiation [2] is presented as an application of CTSVs. The techniques presented take advantage

of the new template features present in the current draft standard of C++ [3], but are not yet implemented in all compilers. A more detailed discussion can be found in [4].

2 Class Templates for Compile-Time Sparse Vectors

A vector is called sparse if only a few of its elements are nonzero. In sparse storage schemes only the nonzero elements are stored, along with some auxiliary information to determine the logical positions of the elements in the vector. Hence space is saved and computational speed gained since some elementary operations are performed only on nonzero elements. An arbitrary sparse vector can be written as a set of value-position pairs

$$\boldsymbol{x} = (< x_{i_1}, i_1 >, < x_{i_2}, i_2 >, \ldots, < x_{i_n}, i_n >).$$

With this notation we can write the sum of two sparse vectors $(1, 0, 1, 0, 0)$ and $(0, 0, 2, 0, 2)$ as

$$(< 1, 1 >, < 1, 3 >) + (< 2, 3 >, < 2, 5 >) = (< 1, 1 >, < 3, 3 >, < 2, 5 >).$$

As can be seen, instead of five additions between the elements, only one addition and two value copy operations are needed to compute the result. To perform only the necessary elementary operations, some kind of indexing scheme must be used to find the right operands. This bookkeeping causes extra overhead which we want to minimise. In CTSVs the bookkeeping can be avoided totally, since it is done by the compiler with no run-time penalty. The indexing information is contained in the template parameters of CTSV classes. In other words, each vector with a different sparseness pattern is a type of its own.

Template definitions can become lengthy, so the code in this article is given for floating-point vectors instead of generic vectors. It is straightforward to generalise the class definitions by making the element type a template parameter.

A special representation for a vector element is needed:

```
template <unsigned int N> class Elem {
public:
  Elem<N> (float v) {}
  Elem<N> () {}
};
template<> class Elem<1> {
public:
  float value;
  Elem<1>(float v) : value(v) {}
  Elem<1>() {};
};
```

The class Elem has a template parameter N, and the class definition for any N is a class having no data members, just two constructors. For N=1 a specialisation

is provided containing a data member for storing a value. So Elem<0> is an empty class, and Elem<1> is a class containing a single floating-point value. Other values than 0 or 1 for N are not meant to be used. The default constructor does nothing. The constructor taking a floating-point parameter initialises the element with a value. To give a uniform interface, the class Elem<0> also has a constructor taking a floating-point value, though it performs no action.

The class representing a CTSV is a collection of Elem<N> objects: Elem<1> objects in nonzero positions and Elem<0> in zero positions. With this kind of element type definitions, we are able to store empty classes as the zero elements. Note however that an object of an empty class may not be totally empty. It is common for a single unused byte to be allocated.

The sparseness pattern of a vector $\boldsymbol{x}$ can be characterised with a bit sequence $\boldsymbol{b}_{\boldsymbol{x}}$, where 1 corresponds to a nonzero element and 0 to a zero element. Since integral types can be manipulated as bit patterns in C++, an unsigned integer template parameter can be used to represent the bit sequence in the generic CTSV class definition. The bit pattern of this template parameter determines the nonzero positions of the CTSV. The class definition is:

```
template <unsigned int N> class CTSV {
public:
  Elem< N&1 > head;
  CTSV< N>>1 > tail;
  CTSV<N>(float v) : head(v), tail(v) {}
  CTSV<N>() {};
  CTSV< N>>1 >& GetTail() { return tail; }
};
template<> class CTSV<0> {
public:
  Elem<0> head;
  CTSV<0>(float v) : head() {};
  CTSV<0>(){};
  CTSV<0>& GetTail() { return *this; }
};
```

The definition is recursive. The head member holds the value of an element. Whether it will be of type Elem<0> (zero element) or Elem<1> (nonzero element) is determined by the least significant bit of the template parameter N. This can be examined by taking the bitwise AND operation with 1. The tail member holds the remaining part of the CTSV. The value of the template parameter for the next step of the recursion is given by right-shifting N one bit. This will eventually result in the template parameter being zero, and the specialisation for $N = 0$ ends the recursion.

The default constructor is defined to do nothing. In addition, a constructor taking a single float argument is provided for initialising a vector with a given value. It will pass the same argument to the head and tail members. In the case of Elem<1> type head, the value is stored, otherwise nothing is done, since the respective Elem<0> constructor is empty. The recursion is ended with the empty

constructor of CTSV<0>. Since CTSV<0> objects have no tail member, GetTail functions are provided. They are needed in the operator definitions to allow uniform access to the vector tail. We may need to get the tail of a CTSV<0> object, and a tail of a tail of a CTSV<0> object, and so on.

The above class definition generates many function calls. Thus it is crucial that all functions be inlined, so empty functions are discarded from the compiled code.

2.1 CTSV Operations

The most common mathematical operations defined for vectors (addition, subtraction, unary negation, multiplication by a scalar, and dot product) can be implemented easily. The definition of addition for CTSVs is given below. Consider two sparse vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ with bit sequences $\boldsymbol{b_x}$ and $\boldsymbol{b_y}$. Vector $\boldsymbol{x} + \boldsymbol{y}$ has then characteristic bit sequence $\boldsymbol{b_x}$ OR $\boldsymbol{b_y}$. In C++ this is:

```
template <unsigned int N, unsigned int M>
inline CTSV<N|M> operator+(const CTSV<N>& a, const CTSV<M>& b) {
  CTSV<N|M> c; plus<N,M>::add(a,b,c); return c;
};
```

This template can be instantiated with two CTSVs having arbitrary bit sequences. The resulting type is a CTSV having a characteristic bit sequence formed by bitwise OR. The operator+ serves as an interface to the actual addition operation implemented as a static member function add of a generic class plus. The template parameters of the plus class are the characteristic bit sequences of the operands of the addition. The code for the plus class is:

```
template<unsigned int N, unsigned int M> class plus {
public:
  static inline void
  add(const CTSV<N>& a, const CTSV<M>& b, CTSV<N|M>& c) {
    add(a.head, b.head, c.head);
    plus< N>>1, M>>1 >::add(a.GetTail(), b.GetTail(), c.GetTail() );
  }
};
template<> class plus<0,0> {
public:
  static inline void add(const CTSV<0>& a, const CTSV<0>& b,
  CTSV<0>& c){}
};
```

In the body of the operator+, an object of the resulting CTSV type is created and passed to the function plus<N,M>::add along with the vectors to be added. This function adds the heads of the vectors with the add function of the Elem<N> classes (defined below) and calls the plus<N>>1 ,M>>1 >::add function recursively with the tails of the operands. The template parameters are shifted right

during the recursion, leading eventually to a call to plus<0,0>::add, which ends the recursion. The add functions for the Elem classes are:

```
inline void add(const Elem<0>& a, const Elem<0>& b, Elem<0>& c){}
inline void add(const Elem<0>& a, const Elem<1>& b, Elem<1>& c){c.v=b.v;}
inline void add(const Elem<1>& a, const Elem<0>& b, Elem<1>& c){c.v=a.v;}
inline void add(const Elem<1>& a, const Elem<1>& b, Elem<1>& c)
                {c.v=a.v+b.v;}
```

The resulting type is "promoted" from Elem<0> to Elem<1> if an Elem<1> type object is involved in the operation. In this way the types are handled correctly. It is easy to see that the compilation of these definitions yields optimal code: for addition of two Elem<0> objects, no code is produced, the addition of Elem<0> and Elem<1> results in a single move, and the addition of two Elem<1> objects generates a single addition.

2.2 Generated Code

To clarify how the above works, we examine the previous example more closely. Vectors $\boldsymbol{x} = (< 1, 1 >, < 1, 3 >)$ and $\boldsymbol{y} = (< 2, 3 >, < 2, 5 >)$ can be constructed and added with the code: CTSV<5> x(1); CTSV<20> y(2); x+y; Note that $5 = 00101_2$ and $20 = 10100_2$. Compilation[1] with Borland C++ 5.01 for Intel 80486 using no optimisation resulted the assembly language code:

```
mov dword ptr [ebp-12],1 // create x
mov dword ptr [ebp-7],1
mov dword ptr [ebp-174],2 // create y
mov dword ptr [ebp-169],2
mov eax,dword ptr [ebp-12] // x+y: 1st
mov dword ptr [ebp-192],eax
mov edx,dword ptr [ebp-7] // 3rd
add edx,dword ptr [ebp-174]
mov dword ptr [ebp-187],edx
mov ecx,dword ptr [ebp-169] // 5th
mov dword ptr [ebp-182],ecx
mov eax,dword ptr [ebp-192] // ret
mov dword ptr [ebp-208],eax
mov edx,dword ptr [ebp-187]
mov dword ptr [ebp-203],edx
mov ecx,dword ptr [ebp-182]
mov dword ptr [ebp-198],ecx
```

The initialisations are the two inevitable move commands for x an y, both having two nonzero elements. Two moves (1st and 5th position) and one addition (3rd) are needed to carry out x + y, which can be seen from the resulting code. The last six lines of the assembly language code originate from the return statement and the implicit invocation of the copy constructor. A clever compiler may avoid this by creating the object directly on the caller's stack. Even without a compiler supporting this *named return-value optimisation* technique, the extra copy can be avoided if we content ourselves with a bit more awkward syntax and use the add function of the plus class directly.

[1] Template operator+ was instantiated manually due to limited template support.

3 An Application: Automatic Differentiation

As an alternative to symbolic calculation of derivatives or using approximate difference values, *automatic differentiation* can be used to obtain derivatives. The derivatives are computed using the well-known chain rule. The function value and the derivatives are evaluated simultaneously with the same expression, but instead of scalars we compute using *automatically differentiable numbers* (ADN). These are objects consisting of a function value and a vector of partial derivatives at the same point. To implement the method, we code the differentiation rules for elementary mathematical operations. In C++ this is done by overloading these operations for ADN objects. There are several texts describing automatic differentiation [2, 5] and also software packages available [6, 7].

The derivative vectors of ADNs are usually sparse. Typically there is only one nonzero derivative at the leaves of an expression tree. When approaching the root, the derivative vectors become more dense. CTSVs are ideally suited for ADN derivative vectors.

A class definition for ADNs using CTSVs as the derivative vectors can be easily constructed. Two data members are needed: a floating-point number for the value of the ADN, and a CTSV for the derivatives. So an ADN class is also generic and shares the template parameter of the derivative CTSV. The overloading of elementary mathematical functions and operators for ADN objects is also simple, with vector addition and scalar multiplication of CTSVs defined. To use ADNs in expressions, a certain element position i is chosen for each variable. Then the characteristic bit sequences having only the ith bit set corresponds to the ith variable. The expression is written using ADN objects with these bit sequences: ADN<1>, ADN<2>, ADN<4> and so forth.

Since CTSVs have no run-time penalty for indexing, ADN expression calculations can be further improved. At the leaf level of the expression tree, the derivatives are either 0 or 1, leading to many multiplications by 1. These can be avoided by keeping track of the positions of 1's. Instead of having zero and nonzero elements, we then have zero, unity and arbitrary elements. There is of course no point in tracking the unity elements at run time just to replace a few multiplications with value moves. But since with CTSVs the tracking is done at compile time, it is perfectly feasible and will in some cases produce significant savings in computation time. Since we now distinguish between three types of elements, there must be three specialisations of the Elem class, so two bits of the characteristic bit sequence of the CTSV are needed for each element. See [4] for C++ code and a more detailed discussion about CTSVs in automatic differentiation.

4 Test Runs

The speed of CTSV operations was compared with ordinary dense vectors (regular C arrays) and sparse vectors with indexing performed at run time (vectors of value/index pairs). The speeds of the addition operations were measured with

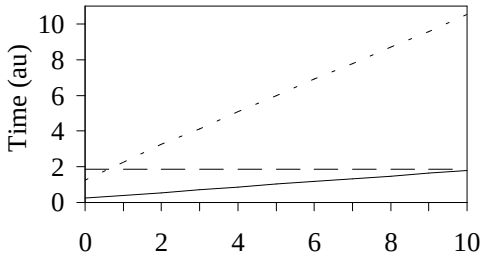


Fig. 1. CPU Time of Vector Addition. *Solid line:* CTSVs; *Dashed line:* dense vectors; *Dotted line:* run-time sparse vectors. Number of nonzero elements is on the x-axis.

10-element vectors, and the number of nonzero elements ranged from 0 to 10. The code was compiled with Borland C++ 5.01 for Intel Pentium processor and optimised for speed. Since the optimiser could not perform return-value optimisation, we used the more awkward syntax to avoid the extra copy constructor invocations. The sparse vectors were allocated statically. Consequently the run-time penalty originates only from indexing, not from memory allocation. The addition functions were called in a loop, and the operand parameters were passed by reference. The CTSV addition for 0 nonzero elements yielded no code. Hence the cost of the function call and looping should approximately equal the cost of the `CTSV<0>` case. The results are shown in Fig. 1.

The speed of ADN expressions was compared to manually optimised symbolically differentiated code. The ADN derivative vectors were implemented as CTSVs (using the refinement described above), as ordinary dense vectors and as ordinary sparse vectors. Results of the test runs are shown in Fig. 2.

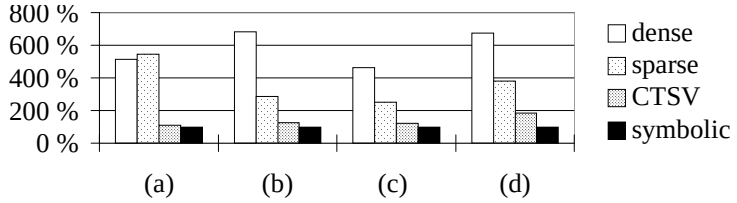


Fig. 2. Relative CPU Time (symbolic = 100%) for Computing Expressions and their Derivatives. (a,b): a multiplication of five variables differentiated with respect to all variables. (c,d): $\sum_{i=1}^{3} a_i e^{b_i t}$ differentiated with respect to a_i and b_i. (a,c): `double` variables. (b,d): `complex` variables.

5 Discussion

A collection of classes for representing sparse vectors in C++ was described. The sparseness pattern of the vector must be known at compile time. For special applications where this restrictive precondition is met, very efficient code can be generated. It was shown how common vector operators can be overloaded for the presented CTSV classes to generate this minimal code from abstract vector expressions.

The classes rely heavily on C++ templates, especially on recursive definitions with non-type template parameters. The sparseness pattern of each vector is represented as a template parameter, i.e. as part of the type information. The sparseness pattern changes in vector operations. Each operation may potentially produce a new vector type. These new template instances are automatically generated from the vector templates by the compiler when encountered. Some of the template features used are quite new and not yet available at all compilers. The features are however part of the standard proposal for C++ [3].

Execution of selected vector operations was compared with dense vectors and ordinary sparse vectors. The speed of CTSVs outperforms both alternatives. The execution speed depends to some extent on the compilers optimisation capabilities; for best results the compiler should be capable of return-value optimisation.

Automatic differentiation was presented as an application of CTSVs. It was shown how to define template classes for automatically differentiable numbers using CTSVs as the derivative vectors. Speed of evaluation for some common expressions was compared with ADNs having alternative vector representations. With our examples, ADNs with CTSV derivatives required CPU time ranging from 20% to 50% of the time used by ADNs with alternative vector representations. The execution speed was only slightly slower than the speed of the symbolically differentiated code.

References

1. Veldhuizen, T.: Using C++ template metaprograms. C++ Report **7** (1995) 36-43.
2. Rall, L. B.: Automatic Differentiation: Techniques and Applications. Lecture Notes in Computer Science **120** (1981) Springer-Verlag, Berlin.
3. 1997 C++ Public Review Document: Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++. ANSI X3J16/96-0225 ISO WG21/N1043.
4. Järvi J.: Compile Time Sparse Vectors in C++. Turku Centre for Computer Science Technical Report No. 107 (1997) (http://www.tucs.abo.fi/publications).
5. Barton, J. J., Nackman L.R.: Scientific and Engineering C++. Ch. 19, Addison-Wesley, Reading Massaschusetts, 1994.
6. Griewank, A., Juedes, D., Utke, J.: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++. ACM Transactions on Mathematical Software **22** (1996) 131–167.
7. Bischof, C. H., Carle, A., Corliss, G. F., Griewank, A., Hovland, P.: ADIFOR: Generating derivative codes from Fortran programs. Sci. Prog. **1** (1992) 1-29.

Compile Time Recursive Objects in C++

Jaakko Järvi
Turku Centre for Computer Science, Finland
jaakko.jarvi@cs.utu.fi

Abstract

This article explores the possibilities of generic programming offered by the template features of C++. We define compile time recursive objects *as instances of class templates which contain other instances of the same template as member variables. With such templates we can define containers that contain objects of arbitrary types, but where the type of each element is known at compile time. The structure of the container is therefore fixed. The technique mimics the polymorphism achieved with dynamic binding and inheritance using static binding and template specialisation. It is obviously less flexible but offers considerable performance gains at runtime.*

We give the template definitions for compile time recursive lists and explain how to perform operations on these lists. As an example application, we use compile time lists in the definition of templates for special sparse vectors and matrices. In these vectors and matrices, the sparseness pattern can be arbitrary but must be known at compile time. The tracking of zero and nonzero elements is performed at compile time. This allows the programmer to use abstract vector and matrix expressions and still reach performance equal to hand-coded operations. This is possible since the compiler can locate the elementary expressions between zero entries and discard them entirely from the resulting code.

Keywords: generic programming, C++ templates, template metaprogramming, sparse matrices

1: Introduction

In C++, generic classes and functions can be written using templates. The template formalism is very rich in the new C++ standard [1] and allows complicated type structures. For instance, compile time bounded loops and branching statements can be written using recursive template definitions and partial specialisation of class and function templates. Template usage of this type is called *template metaprogramming*. This novel technique has been previously used to speed up algorithms by *program specialisation* [2, 3], calculate binary constants at compile time [4] and define portable integral types [5].

In this article the idea of template metaprogramming is developed further and applied to *compile time recursive objects*. By compile time recursive objects we mean template classes which have other instances of the same template class as data members. Such objects are not so rare, e.g., a two-dimensional array can be implemented as a generic array class, where the elements themselves are arrays. Recursive tree-like objects are encountered in *expression*

This work was supported by the Academy of Finland, grant 37178.

templates [6], which are templates representing expression trees. This technique aims at avoiding the creation of temporary vector objects in the evaluation of vector expressions.

This article examines the nature of compile time recursive objects more profoundly. We define templates which can be used as containers where the type of each element can be arbitrary but statically bound, being thus fixed at compile time. We show how to perform operations with them.

The most basic construct is a recursive list class, which is discussed in detail. We show the template definitions and describe how different operations for these types can be written. From a general approach we move to a specific application. We show how efficient operations for sparse vectors and matrices can be written using compile time recursive lists and template metaprogramming.

The foremost benefits achieved with the techniques presented are evidently computational speed and, to some extent, type safety. The concept of compile time recursive objects as such is intriguing and needs to be explored more.

At the moment, some of the template features used in this article are not implemented in all of the currently available C++ compilers. All features are, however, part of the new standard for C++ and will therefore most likely be implemented by major compiler vendors soon.

1.1: Template metaprogramming

As an introduction to a reader not familiar with the newest C++ template features, we briefly describe the concept of template specialisation and give an example of a simple template metaprogram. Using template specialisation, partial or total, alternative definitions for a *primary* template class can be given. During the instantiation of a template, the types of the template parameters determine whether the instance is generated using the primary template or one of the specialisations. The template arguments are matched with the template argument lists of each specialisation and the most specialised definition is selected. This is done according to a set of partial order rules. Note that the selection must be unambiguous. Basically the same mechanism applies both to class and function templates. In function templates the template arguments can be deduced from the function arguments.

Template metaprograms utilise template specialisation. The primary template is defined as recursive and specialisations are used to end the recursion. As an example of a simple template metaprogram, consider the following code calculating the factorial at compile time:

```
template<int N> struct F { enum { val = N * F<N-1>::val }; };
template<> struct F<0> { enum { val = 1 }; };
```

In the primary template, the enumeration constant val is intended to contain the factorial of the template parameter N. In the definition of the constant val of the class F<N>, the constant val of the class F<N-1> is used. Hence, the instantiation of F<N> triggers recursively the instantiation of F<N-1>. A specialisation for F<0> is given to end the recursion. All this occurs at compile time and val gets the value N! at compile time.

The above is an example of compile time recursion with respect to integral template parameters. In the remainder of this article, the recursion takes place with respect to normal type parameters.

2: Compile time lists

Consider the traditional LISP view of a list as a record consisting of a head and tail, which is itself a list. Such a list can be written with the following template definitions:

```
template <typename HT, typename TT>
class list {
public:
  typedef HT head_type;
  typedef TT tail_type;

  head_type head;
  tail_type tail;
};

class nil() {};
```

The list contains only two member variables, head and tail. The typedefs are provided to allow reference to the types of the variables in a uniform way. The definitions allow considerable freedom in instantiation, as HT and TT can be of arbitrary types. The template is, however, not intended to be instantiated arbitrarily. HT can be an arbitrary type but TT should be another instance of the list template or the nil class, which is the end mark of the recursive list. It would be easy to constrain the template to allow only such instantiations where the tail is of type list or nil. This would, however, complicate the syntax and for simplicity is not shown.

Note that the elements of the list may be of arbitrary types, so the class is a *heterogeneous statically bound container*. As an example consider the following:

```
class A; class B; class C;
list<A, list<B, list< A, list< C, nil >>>> x;
```

According to this definition, x is an object which has objects of types A, B, A, C and nil as nested member variables. These variables can be accessed using the common notation (e.g. x.tail.tail.head) but also using generic functions and classes. E.g., the nth item of the list can be accessed at compile time, a certain operation can be performed on each element in the list and generic operations between these structures can be defined. To clarify how compile time lists can be manipulated, we give two examples. The first shows how to perform a certain operation for each object in the list and the second describes the concatenation of two compile time lists.

2.1: Operations of compile time lists

Let us assume that each element class X has a function void apply(X&) defined, which performs some action on the element. Each element is passed to the correct apply function with a call to the following function:

```
template<typename T> inline void traverse(T& a) {
  apply(a.head);
  traverse(a.tail);
};

void traverse(nil&) {};
```

Note that this seemingly recursive function is not really recursive. Although it calls a traverse function, it is another instance of the same function template and thus a different function.

This is the basic scheme for processing objects in the list. Operations can be more complicated and they may potentially produce new instances of the list template, i.e., new types. These types must be expressed using the template parameters of the generic function. This is achieved by using a C++ idiom known as *traits* [7]. Note that these trait templates can also be recursive. The concatenation illustrates this point:

```
template<typename L1, typename L2>
inline type_concat<L1, L2>::t concat(const L1& list1, const L2& list2) {
  type_concat<L1, L2>::t result;
  tr_concat(list1, list2, result);
  return result;
};
```

This is the interface function, which creates an object to be returned, passes it along the input parameters to a compile time recursive traversing function tr_concat and finally returns the result. The resulting type is given by type_concat<L1, L2>::t, which is a type definition in a generic class type_concat. The sole purpose of this trait class is to define the type of the concatenation given the types of the lists to be concatenated. The generic class is defined as

```
template<typename L1, typename L2>
struct type_concat {
  typedef list<L1::head_type, type_concat<L1::tail_type, L2>::t> t;
};

template<typename L2> struct type_concat<nil, L2> { typedef L2 t;}
```

The type of the concatenation of two lists is thus a list, where the head is of the same type than the head of the first list. The type of the tail is given by concatenating the tail of the first list with the second list recursively. The partial specialisation is for ending the compile time recursion in the case where the first list is empty.

To traverse the lists and perform the actual concatenation, a template function and two partial specialisations are needed:

```
template<typename L1, typename L2, typename L3>
inline void tr_concat(const L1& a, const L2& b, L3& result) {
  result.head = a.head;
  tr_concat(a.tail, b, result.tail);
};

template<typename L2, typename L3>
inline void tr_concat(const nil& a, const L2& b, L3& result) {
  result.head = b.head;
  tr_concat(a, b.tail, result.tail);
};

template<typename L3> inline void tr_concat(const nil&, const nil&, L3&) {};
```

The given templates for lists and list manipulation illustrate the basic mechanisms of

class definitions and operations for compile time recursive objects. To be feasible, the usage of these templates must be reasonably convenient. The programmer should not have to write such declarations as list<x, list<y, ... >> or access elements using such notation as aList.tail.tail.tail.head. For better syntax, special templates for element access and initialisation of objects are needed, even macro definitions may prove useful. Since these needs and possibilities may be application dependent, they will be discussed more with the matrix example below.

2.2: About compilation

In order to understand the potential benefits offered by the above described techniques, some issues concerning compilation and optimisation must be discussed. One can notice that traversing the list structures produces many functions calls. These functions are, however, very small. They commonly perform some operation on the list heads and pass the tails to another function. This means that the functions can be effectively inlined and thus their overhead eliminated. With this in mind, it can be seen that traversing these compile time structures and accessing their elements include no runtime cost. Inlined traversal functions produce unrolled loops when compiled and only the operations performed on the individual elements of the lists yield any code. Another important aspect is the static binding of the elements. Though the elements in the container can be of arbitrary types, the functions called for each object in the container are bound statically. This makes the inlining possible here as well. To illustrate the use of the recursive compile time objects, we proceed with a concrete example concerning sparse vectors and matrices.

3: Sparse vectors and matrices as compile time recursive objects

A vector or matrix is called sparse, if a considerable degree of its elements are zero. The sparseness of vectors and matrices can be exploited when performing computations with them. Vector and matrix operations can be defined to perform only those elementary operations which yield nonzero results. To exploit the sparseness, some indexing scheme must be used to find the right operands for the elementary operations. This bookkeeping causes extra overhead. If the vectors and matrices are small or contain relatively many nonzero elements, sparse techniques may become costly due to this overhead. As an example, consider the sparse vectors $a = (1, 0, 1, 0, 0)$ and $b = (0, 0, 2, 0, 2)$ and let $c = a + b$. To calculate the resulting vector $c = (1, 0, 3, 0, 2)$, only one addition and two value copy operations are needed. However, due to the indexing overhead, it would probably be much faster to perform all five additions than to apply some sparse scheme.

If the sparseness patterns of the vectors and matrices are known beforehand, the programmer may eliminate the indexing cost by writing specialised code which performs only the necessary operations. Hence the programmer in effect performs the indexing himself. This is of course very tedious and error-prone. Using compile time recursive objects the sparseness pattern can be expressed in the type of a vector or matrix. It is then possible to make the compiler generate this specialised code from abstract vector and matrix operations and thus avoid the bookkeeping cost entirely. To accomplish the task, we first define template classes for *compile time sparse vectors*.

3.1: Compile time sparse vectors

Compile time sparse vectors (CTSV) are basically compile time lists and can be derived from the list template classes given above:

```
template <typename HT, typename TT>
class ctsv : public list<HT, TT> {};

class zero { public: zero(const zero&) {} }
```

For simplicity, no members are given for the ctsv class here. It would, however, be worthwhile to define various assignment operations and constructors. Objects of the class zero are used as the zero elements in the ctsv. Note that the zero class has a copy constructor defined. Even though the class zero is totally empty, the objects of this class reserve some memory. This is to guarantee that each object has an identity. By defining explicitly the copy constructor to be empty, we prevent the copying of this extra memory, which might be performed by the compiler-generated default copy constructor. With this class definition the creation and copying of the zero objects bears no runtime cost.

With the above definitions, the type of a vector $(1.1, 0, 0, 2.2)$ would be

```
ctsv<double, ctsv<zero, ctsv<zero, ctsv<double,nil>>>>.
```

The type clearly contains the information about the sparseness pattern of the vector.

The intention is that the programmer can write abstract vector expressions with CTSVs and the types are deduced automatically leading to more complex types as the computation advances. To define operations between these vectors, we need to define the resulting type of the operation, how to find the matching elements within each vector, and how to apply the operation between matching elements. We give the definition of the addition operation, since it is illustrative. Other operations can be defined using the same principles.

3.1.1: Addition of two vectors

To define the result type of addition, we deal with the type changes between the element types. Again, this is done with trait classes:

```
template<typename T1, typename T2> struct type_sum {};
template<> struct type_sum<double, double> { typedef double t; };
template<> struct type_sum<double, zero> { typedef double t; };
template<> struct type_sum<zero, double> { typedef double t; };
template<> struct type_sum<zero, zero> { typedef zero t; };
```

These only state that whenever a double is involved in the addition, the result is of type double. The templates could also be defined in a more general fashion using partial specialisation.

The type of the whole addition is defined as a recursive trait class, analogously to the type deduction of the list concatenation:

```
template<typename H1, typename T1, typename H2, typename T2>
struct type_sum<ctsv<H1, T1>, ctsv<H2, T2> > {
  typedef ctsv<type_sum<H1, H2>::t, type_sum<T1, T2>::t> t;
};

template<> struct type_sum<nil, nil> { typedef nil t;};
```

The addition operator is itself just an interface function. It creates the object to be returned, lets the actual work be done by the traversing function add and finally returns the result:

```
template<typename HT1, typename TT1, typename HT2, typename TT2>
inline type_sum<ctsv<HT1, TT1>, ctsv<HT2, TT2> >::t
operator+(const ctsv<HT1, TT1>& a, const ctsv<HT2, TT2>& b) {
  type_sum<ctsv<HT1, TT1>, ctsv<HT2, TT2> >::t result;
  add(a, b, result);
  return result;
};

template<typename V1, typename V2, typename V3>
inline void add(const V1& a, const V2& b, V3& result) {
  add(a.head, b.head, result.head);
  add(a.tail, b.tail, result.tail);
};

template<> void add(const nil&, const nil&, nil&) {};
```

The add functions call the elementary add functions to add the heads and then the tails recursively. The elementary add functions are defined as

```
inline void add (const double& a, const double& b, double& c) { c = a + b; };
inline void add (const double& a, const zero& b, double& c) { c = a; };
inline void add (const zero& a, const double& b, double& c) { c = b; };
inline void add (const zero& a, const zero& b, zero& c) {};
```

These functions obey the type promotions defined with the type_sum traits. It is easy to see that when compiled, these functions yield the minimal instructions for summation. To add two zero objects, no code is produced, the addition of a double and a zero results in a single value copy operation and the addition of two doubles in a single addition. These are the functions which perform the additions optimally, the rest of the definitions are actually only for determining the correct function to be called at compile time.

As an example, consider the previous vectors a, b and c. The sparseness patterns of the vectors are $(1, 0, 1, 0, 0)$, $(0, 0, 1, 0, 1)$ and $(1, 0, 1, 0, 1)$, respectively. The following assembly language code is the result of the compilation of the statement c = a + b using KAI C++ 3.2.d compiler under Linux for Intel Pentium processor.

```
movl    -16(%ebp),%eax      // move 1st element of a
movl    %eax,-64(%ebp)
flds    -8(%ebp)            // add the 3rd elements
fadds   -32(%ebp)
fstps   -56(%ebp)
movl    -24(%ebp),%eax      // move the 5th element of b
movl    %eax,-48(%ebp)
```

As can be seen, the code is minimal, performing only the elementary operations needed without any control instructions.

3.2: Compile time sparse matrices

A matrix is a vector of vectors, hence we can use the ctsv template for matrices as well. E.g., the matrix

$$\begin{pmatrix} a & 0 & b & 0 \\ 0 & c & 0 & 0 \\ d & 0 & e & 0 \\ 0 & 0 & 0 & f \end{pmatrix}$$

is of type

```
ctsv< ctsv< double, ctsv< zero,   ctsv< double, ctsv< zero,   nil >>>>,
ctsv< ctsv< zero,   ctsv< double, ctsv< zero,   ctsv< zero,   nil >>>>,
ctsv< ctsv< double, ctsv< zero,   ctsv< double, ctsv< zero,   nil >>>>,
ctsv< ctsv< zero,   ctsv< zero,   ctsv< zero,   ctsv< double, nil >>>>,
nil >>>>
```

The previously defined vector operations can be used directly. For instance, the addition was defined to first add the heads and then the tails recursively. Now the addition of heads is a vector operation instead of a scalar operation and it is performed by adding first the heads and then the tails recursively. New operations, such as matrix transpose and multiplication, must of course be defined. The type deduction classes tend to become rather complicated and they are not shown here.

3.3: Declaring types and initialising variables

The vector and matrix types are rather tedious to write as such. Various helper templates can be defined to ease this task. We can achieve such notation as V<1,0,1,0,0>::t for defining vector types and

```
typedef M< V<1, 0, 1, 0>::t,
           V<0, 1, 0, 0>::t,
           V<1, 0, 1, 0>::t,
           V<0, 0, 0, 1>::t>::t matrix_type_1;
```

for matrix types. The syntax is fairly reasonable. The assignment of values to vectors and matrices can also be simplified by using the operator overloading capabilities of C++. The comma operator can be overloaded to provide a means to initialise the vectors and matrices with values.

```
matrix_type_1 m; double a, b, c, d, e, f; zero z;
m =   a,   z,   b,   z,
      z,   c,   z,   z,
      d,   z,   e,   z,
      z,   z,   z,   f;
```

This is quite convenient, particularly since an attempt to set a zero to a nonzero position or vice versa results in an error at compile time. For vectors the initialisations are as effective as normal assignments to named variables. For matrices there may be some overhead. It is, however, easy to initialise matrices rowwise.

Usually the sparseness patterns of the initial vectors or matrices are simple and more complicated types result from expressions between them. These complicated types need

not be specified, since they are automatically deduced. A good example of this is *automatic differentiation* [8]. It is a technique for computing derivatives without the need to symbolically differentiate the function, but avoiding the use of divided difference approximations as well. CTSVs have proved useful in this application as shown in [10, 11], where they are used to represent the vector of partial derivatives in automatically differentiable numbers. These vectors are initially canonical unit vectors, consequently their types are easy to define. These vectors are used as variables in mathematical expressions. During the evaluation several intermediate temporary vectors and finally a dense vector are produced. The intermediate vectors may have arbitrarily complex sparseness patterns, but as pointed out, they are automatically deduced and need not be specified.

3.3.1: Transformation matrix example

As an example of the usage of CTSVs, consider the type matrix_type_1 defined above. It defines the sparseness pattern of a transformation matrix in the homogeneous coordinate system. When a vector is multiplied with the matrix

$$\begin{pmatrix} cos(\theta) & 0 & -sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ sin(\theta) & 0 & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

it is rotated around the y-axis [9]. Using CTSVs, a subroutine to perform this transformation can be written as follows:

```
void rotate1(double theta, const dense_vec<4>::t& v, dense_vec<4>::t& result) {
  matrix_type_1 m; zero z;
  double cost = cos(theta), sint = sin(theta);

  row<0>::get(m) = cost,   z,   -sint,   z;
  row<1>::get(m) = z,      1,   z,      z;
  row<2>::get(m) = sint,   z,   cost,   z;
  row<3>::get(m) = z,      z,   z,      1;

  result = v*m;
}
```

dense_vec<4>::t gives the type of a dense four-element vector, and the odd-looking code row<N>::get(m) is just an invocation of a template metaprogram to get the Nth row of the matrix m for rowwise initialisation.

Despite the syntactical differences, the code has apparent correspondence with the mathematical notation of the transformation. This is not the case for a low-level implementation of the same transformation, where the matrix multiplication is decomposed into elementary operations by hand:

```
void rotate2(double theta, double v[ ], double result[ ]) {
  double cost = cos(theta), sint = sin(theta);

  result[0] = v[0] * cost + v[2] * sint;
  result[1] = v[1];
  result[2] = v[2] * cost - v[0] * sint;
  result[3] = v[3];
}
```

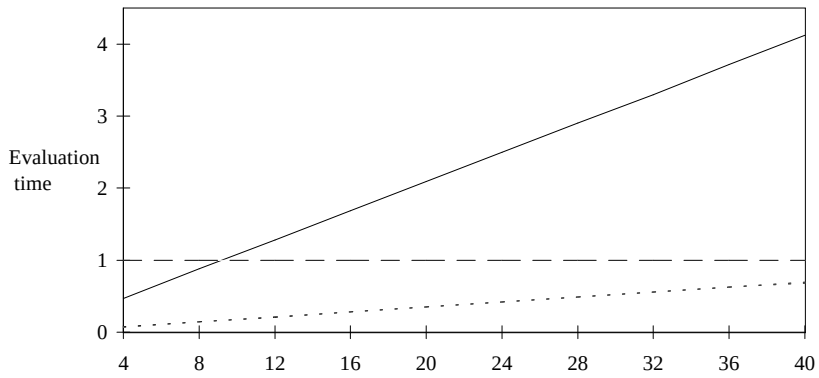


Figure 1. The relative evaluation time of the dot product of two vectors of length 40; *dotted lines:* **CTSVs and hand-coded C (they overlap entirely);** *dashed line:* **dense vectors;** *solid line:* **runtime sparse vectors. The number of nonzero elements is shown on the x-axis.**

One might suspect that the low-level implementation would be considerably faster, but when compiled (again with KAI C++, optimisation flags +K3 -O2), the CTSV implementation produced 45 sequential machine instructions compared with the 44 instructions of the low-level code. Further, if the traversing function was called directly (replacing the last line `result = v*m;` of `rotate1` with the line `multiply(v,m,result);`) the assembly listings were exactly identical.

3.4: Speed comparisons

To demonstrate the attainable performance gains of CTSVs, we measured the computing time of a dot product of two double vectors (Fig. 1). The length of the vectors was 40 elements, and the number of nonzero elements was increased from 4 to 40 with step size 4. The nonzero elements were in the same positions in both vectors, so the number of multiplications needed was the same as the number of nonzero elements. We used four different vector representations: ordinary dense vectors, ordinary sparse vectors, CTSVs and hand-coded C. In the ordinary dense vectors the dot product is a for loop from 1 to 40, so the cost is not dependent on the number of zeroes. The ordinary (runtime) sparse vectors were implemented as an array of values and a corresponding array of indices. The CTSVs were as described in this article, and finally in the hand-coded C version, each elementary operation was written directly, so the loop was totally unrolled by hand and only the nonzero elementary operations were computed. To our knowledge, the hand-coded C was the fastest way to perform the dot product without resorting to any processor-dependent tricks or inline assembly code.

The test was compiled with the KAI C++ 3.2.d compiler using optimisation flags +K2 and -O2 and it was run under Linux operating system on a Pentium PC. The test shows that the normal sparse techniques become unattractive quite soon as the number of nonzero entries increases. The execution speed of the CTSV dot product and the hand coded C version was almost identical in all cases. This was evident since the assembly code they produced was essentially the same.

It can also be seen that the unrolled versions were somewhat faster than the for loop implementation even when the vectors had no zeroes. This lead vanishes if the vector sizes

are increased to the point where the code no longer fits in the processor cache.

In [10, 11], more execution speed comparisons between CTSVs, runtime sparse vectors and dense vectors can be found. The results were very favourable for CTSVs. Also, the speed of automatic differentiation of various expressions was compared using different implementations of the partial derivative vector. CTSVs outperformed clearly other alternatives and came very close to the speed of symbolically differentiated codes. The implementation of CTSVs was slightly different but the assembly code produced was similar, so the results are applicable here as well.

4: Restrictions in compilation

Since the template definitions of compile time recursive objects can be deeply recursive, there will of course be some practical limitations to the depth of the recursion set by the compiler. Some compilers may have strict limits for the number of nested recursive template instantiations, but several compilers provide a parameter for controlling this, allowing thus arbitrarily many nested instantiations. At some point, available resources, such as the amount of memory or the compilation time needed, set limits to this technique.

However, our experiments show that the technique can be applied to reasonably sized objects and compiling is fairly rapid. In the case of CTSVs, the compilation of a 100-element dot product succeeded with no difficulties. The compilation of matrix operations is somewhat harder since the type deductions are more complicated. However, as CTSV operations produce a block of sequential (unrolled and inlined) code, it is hardly desirable to apply the technique for very long vectors or large matrices.

5: Conclusions

This article presents compile time recursive objects as a new technique for generic programming in C++. The technique relies heavily on partial template specialisation and recursive template definitions. It is shown how to define a compile time list structure, and how to perform operations on it. With this template the programmer can define special heterogeneous container types which may contain objects of arbitrary types and yet the types are statically bound. Despite this, the elements of these containers can be accessed and operated on in a uniform manner, and due to the static binding, also very rapidly. The application of a certain template function to each element of such container has no runtime cost of traversing through the collection. The cost originates only from the operations performed on each element.

This article shows how template classes for special sparse vectors and matrices can be defined as compile time recursive objects. In these vector objects, the type includes the information about the positions of the nonzero elements. This makes it possible for the compiler to decide which elementary operations must be performed and which yield a zero element and thus can be discarded. It is shown that with compile time recursive objects, the programmer can write abstract vector and matrix operations and does not lose anything on performance compared with coding all elementary operations by hand. In our examples, the compiler generated assembly code essentially equivalent to the hand-coded alternative.

The concept of compile time recursive objects is attractive. The technique can be seen as an imitation of the polymorphism achieved with dynamic binding and inheritance, by

static binding and template specialisation. Since the main advantage offered by this is speed, scientific computing is the potential application area for compile time recursive objects. This article demonstrates the usefulness of the technique in the implementation of special sparse vectors and matrices.

References

[1] ISO/IEC 14882 Standard for the C++ Programming Language.

[2] Veldhuizen, T.: Using C++ template metaprograms, C++ Report **7** no. 4 (1995) 36-43.

[3] Veldhuizen, T.: Linear algebra with C++ template metaprograms, Dr. Dobb's Journal **21** no. 8 (1996) 38-44.

[4] Pescio, C.: Binary Constants using Template Metaprogramming, C++ Users Journal, February (1997).

[5] Pescio, C.: Template Metaprogramming, C++ Report **9** no. 7 (1997) 22-27.

[6] Veldhuizen, T.: Expression Templates, C++ Report **7** no. 5 (1995) 26-31.

[7] Myers, N. C.: A new and useful template technique: "traits", C++ Report **7** no. 5 (1995) 32-35.

[8] Rall, L. B.: Automatic Differentiation: Techniques and Applications, Lecture Notes in Computer Science **120** (1981) Springer-Verlag, Berlin.

[9] Mortenson M. E.: Geometric Modeling, (1985) Wiley, New York.

[10] Järvi J.: Processing Sparse Vectors during Compile Time in C++, Scientific Computing in Object-Oriented Parallel Environments, Lecture Notes in Computer Science **1343** (1997) 41-48 Springer, Berlin.

[11] Järvi J.: Compile Time Sparse Vectors in C++, Turku Centre for Computer Science, Technical Report No. 107 (1997) (www.tucs.fi/publications).

Acta Cybernetica **14** (1999) 1-.

Object-Oriented Model for Partially Separable Functions in Parameter Estimation*

Jaakko Järvi†

Abstract

In parameter estimation, a model function depending on adjustable parameters is fitted to a set of observed data. The parameter estimation task is an optimisation problem, which needs a computational kernel for evaluating the model function values and derivatives. This article presents an object-oriented framework for representing model functions, which are partially separable, or *structural*. Such functions are commonly encountered, e.g., in spectroscopy.

The model is general, being able to cover a range of varying model functions. It offers flexibility at runtime allowing the construction of the model functions from predefined component functions. The mathematical expressions are encapsulated and a close mapping between mathematics and program code is preserved. Also, all interfacing code can be written independently of the particular mathematical formula. These properties together make it easy to adapt the model to different problem domains: only tightly controlled changes to the program code are required.

The paper shows how derivatives of the model function can be computed using automatic differentiation relieving the programmer from writing explicit analytical derivative codes.

The persistence of the objects involved is discussed and finally the computational efficiency of the function and derivative evaluation is addressed. It is shown that the benefits of the object-oriented model, namely the higher abstraction level and increased flexibility, are achieved with a very moderate loss of performance. This is demonstrated by comparing the performance with low-level tailored C-code.

1 Introduction

Even though object-orientation (OO) has become the dominating programming paradigm, it is quite slowly adopted to numerical applications, mainly because of the poor efficiency of OO programs in numerical codes. The progress in programming techniques and compilers is changing this situation and makes it possible to

*This work was supported by the Academy of Finland, grant 37178.

†Turku Centre for Computer Science, Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland, email: jaakko.jarvi@cs.utu.fi

take advantage of OO in numerical codes without a significant performance penalty [16]. This is demonstrated in this paper describing an OO model for parameter estimation of structural, partially separable functions.

The task of modelling data is commonly encountered in numerous application fields. The goal is to fit a model that depends on adjustable parameters to a set of observed data. A cost function, such as the sum of squared differences, is chosen to measure the agreement between the model and data. This function is minimised by adjusting the parameters of the model according to some optimisation algorithm.

The model can be based on some underlying theory about the data or be just a sum of convenient functions, such as polynomials. This article focuses on partially separable model functions, where the function is a sum of *component functions*, e.g., a spectrum consisting of a sum of spectral lines. The OO model presented in this article was developed while working on nuclear magnetic resonance (NMR) spectra estimation. Hence, the article includes a case study of NMR spectral fitting to make the ideas presented more concrete.

Numerous algorithms have been described for model fitting tasks in the literature [2, 14]. They are usually presented from the numerical analysis viewpoint, treating the model as a plain vector of parameters and a function for evaluating values and derivatives. However, this *flat* representation of the model function is not necessarily natural. The model may be structural consisting of several component functions, which possibly correspond to some real life entities. The function representation should be flexible. It should be possible to specify the composition of the component functions at runtime, rather than fix them in the program code. Furthermore, the function representation should be able to handle dependencies between parameters of different component functions. The flat model representation is therefore inconvenient for the user and it is the application developer's task to provide a conversion to and from the structural representation.

This article presents an OO model to serve as an intermediate link between the two representations described above. The model provides simultaneously an efficient computational kernel for the optimisation algorithms and the structured view for the user. It is a collection of classes comprising a core to represent structured model functions. These core classes implement the basic structural and flat views to the model function, as well as the mechanisms for function value and derivative calculations.

The extension of the core model for a specific application is done by providing a simple class for each type of component function. Essentially only member functions specifying the mathematical formulae of the component functions are required in these classes. Consequently, the particular mathematical expressions are encapsulated and the mathematical structures of the problem domain are preserved in the program code. This means that the necessary changes to program code are minor and well controlled if the model is applied to a different application area.

The model utilises the concept of *automatic differentiation* [15] for derivative computations. This relieves the programmer from writing analytical derivative codes. Automatic differentiation is made transparent to the programmer with operator overloading.

The core classes implement all the functionality needed for constructing component functions and their parameters. The user interface for this task can therefore be built solely based on the core classes. The addition of new classes to the model hierarchy does not cause any need for changes in the interfacing code. In section 3.5 we give an example of a user interface built in this manner.

This paper also discusses the computational efficiency and shows that the overhead arising from the higher abstraction level and greater runtime flexibility of the OO model is very moderate compared with a low-level C-code implementation. Persistence, i.e., the ability to store and retrieve the objects of the model is also considered.

The crucial parts of the model are presented using C++ language, but the model can be implemented in any language supporting inheritance, dynamic binding and operator overloading. However, the test runs were performed using a C++ implementation.

There are few descriptions of using object orientation together with parameter estimation in the literature. Related work can be found from [11, 17] containing general descriptions of computer systems sharing some similarities with our model. For description of an NMR analysis software built using a variant of the object oriented model presented here, see [10].

2 Parameter estimation problem

The task of fitting a parametric model function to a set of observed data points can be seen as minimisation of a cost function describing the distance between the model and the data. A common choice for the cost function is the sum of squares function. This least-squares model fitting problem can be stated as follows:

Let $y(x_i), i = 1, \ldots, m$ be a set of observed data points, $\mathbf{p} = (p_1, \ldots, p_k)$ be a vector of model parameters and $\hat{y}(x, \mathbf{p})$ a parameter-dependent model function. The maximum likelihood estimate of the parameters is obtained by minimising the chi-square function

$$\chi^2(\mathbf{p}) = \sum_{i=1}^{m} \left(\frac{y(x_i) - \hat{y}(x_i, \mathbf{p})}{\sigma_i} \right)^2 , \tag{1}$$

where σ_i is the standard deviation of the measurement error of the ith data point. This formulation leads to a possibly non-linear optimisation problem which can be solved with iterative methods, most commonly with Levenberg-Marquardt or Gauss-Newton algorithms [2, 14]. The idea is to improve iteratively the trial solution

$$\mathbf{p_{new}} = \mathbf{p_{current}} + \mathbf{\Delta p} \tag{2}$$

until an acceptable solution is found. The change $\mathbf{\Delta p}$ is determined using the gradient and usually an approximation of the Hessian of the cost function. These in turn require calculation of the partial derivatives $\frac{\partial \hat{y}(x, \mathbf{p})}{\partial p_s}, s = 1, \ldots, k$ of the model function. Even though each iteration typically involves additional costs,

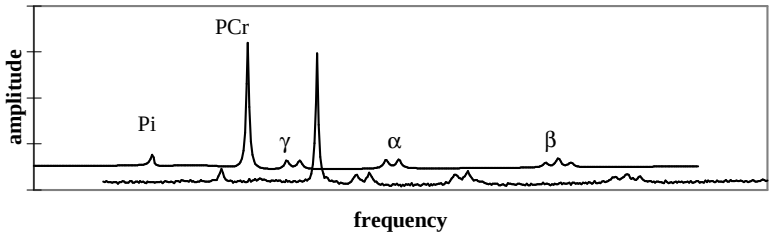


Figure 1: Example of a ^{31}P NMR spectrum (lower curve) and a model function (upper curve) fitted to the spectrum. α, β and γ peak groups originate from ATP molecules. The measured spectrum is shifted rightwards for clarity.

such as solving a linear system of equations, the calculation of the model function and derivative values often dominate the overall cost.

The above clarifies the numerical view to the parametric estimation problem. The algorithms developed for the estimation must be supplied with the parametric model function, functions for the partial derivatives and the vector of modifiable parameters. Furthermore, $\hat{y}(x, \mathbf{p})$ is typically calculated at several points with constant $\mathbf{p}$. In cases we are interested in, $\hat{y}(x, \mathbf{p})$ is partially separable, that is, $\hat{y}$ can be represented as a sum of component functions $\hat{y}_j, j = 1, \ldots, n$, each being dependent on only r_j parameters, where $r_j << k$.

2.1 NMR spectroscopy case

In NMR spectroscopy, a signal of damping oscillations (FID) emitted by certain atomic nuclei (e.g. ^{31}P) is observed. An NMR spectrum is a Fourier transform of this signal. The spectrum contains peaks or resonance lines corresponding to nuclei in various compounds. The amplitude of a single peak is proportional to the number of equivalent nuclei resonating at that frequency. [6]

A typical ^{31}P NMR spectrum is shown in Fig. 1. Signals of inorganic phosphate (Pi), phosphocreatine (PCr) and adenosine triphosphate (ATP) can be identified from the spectrum. The aim is to find the amplitudes and frequencies of the identified compounds. This is done by quantifying the spectrum or the FID, which is represented as a superposition of parametric functions, each corresponding to a single resonance line. This parametric model function is fitted to the measured signal and the results, peak intensities and frequencies, are calculated from the model parameters. Fig. 1 also shows a fitted model.

Basically we have a structural model function consisting of a sum of component functions, the resonance lines. Several lineshapes are encountered, the most common being the Lorenz function described by amplitude A, frequency f, phase ϕ and damping factor d. A model of n reconance lines in a somewhat simplified

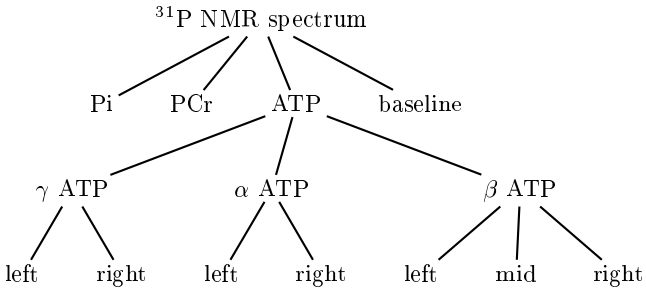


Figure 2: Example of a model function instance.

form in time domain is then

$$\hat{y}(t, \mathbf{p}) = \sum_{j=1}^{n} A_j \cos(2\pi f_j t + \phi_j) e^{-d_j t}, \tag{3}$$

where $\mathbf{p} = (A_1, f_1, d_1, \phi_1, \ldots, A_n, f_n, d_n, \phi_n)$. As can be seen, the sum function is partially separable. Note that, contrary to this simplified expression, the NMR signal can contain different lineshapes and there may be additional terms in the sum. [5]

Dependencies between parameters of different component functions are typical for NMR models. Consider the ATP molecule. It is known a priori that 7 peaks altogether originate from the ATP molecules. The peaks come in three groups: α, β and γ. These groups have equal amplitudes. The groups α and γ consist of two peaks each having again equal amplitudes. The β-group consists of three peaks with relative amplitudes $1 : 2 : 1$. The frequency differences between the peaks inside the groups are known and it is reasonable to assume that the damping factors of all the peaks are equal. Taking these into consideration, the amplitudes, damping factors and frequencies of 7 peaks are actually defined by only one amplitude, one damping factor and three frequency parameters. The hierarchical structure of ATP and other peaks in the NMR example spectrum is depicted in Fig. 2.

To sum up the problem setting, the estimation of the parameters of the function $\hat{y}$ is the task to be performed. This is done by minimising the chi-square error with respect to the measured signal, where the partial derivatives of $\hat{y}$ must be calculated repeatedly. Function $\hat{y}$ has a hierarchical structure corresponding to the peaks in the spectrum.

3 Object-oriented model

Significant savings in development time can be achieved with careful design of the model function representation. In the case of structural model functions, the utmost goal is flexibility. The number and type of the component functions may vary and there may be common or related parameters between the component functions.

The model function representation ought to be able to handle these situations with ease and yet be able to compute the function value and derivatives efficiently.

An important issue is the user interface for managing the model functions. The user constructs the model functions and observes or edits the model parameters. The programmer's task to provide this interface for varying models is considerably alleviated if the interface can be implemented without the need to know the actual types or number of the component functions. The term *user* refers to a human operator of a computer program whereas by *client* we denote the programmer or code calling the functions or using other services of the object-oriented model.

The object-oriented approach provides a convenient means to build a function representation to meet the requirements detailed above. The model consists of two separate class hierarchies, the *function hierarchy* and the *parameter hierarchy*. An essential component is also a library for automatic differentiation. The hierarchies are first discussed accentuating the client view to the classes and then the process of function value and derivative evaluation is clarified. While reading, the reader may consult the object diagram in Fig. 6 representing the NMR example as objects from function and parameter hierarchy.

3.1 Function hierarchy

The classes of the function hierarchy (Fig. 3) represent the component functions of the structural model function (the nodes of the tree in Fig. 2). The base of the hierachy is the abstract base class base_model, which defines the interface for the function classes; each function can compute the value and derivatives at a given point. The base_class maintains a vector of parameters and defines member functions for accessing them. Different component functions are derived from the base_model class. These can be either *elementary* or *composite* functions.

Composite functions maintain a list of other component functions. They simply group other components. A composite function computes its values and derivatives by calling the evaluation functions of its *child* functions. Each composite model *owns* the models in its child list. The top_model class represents the whole model function to be fitted and implements the interface to the client code. It also maintains the vector of the adjustable parameters used by the optimisation algorithm.

The generic elementary_model class encapsulates the common features of the component functions to make the derived classes as simple as possible. The template parameter of the generic class specifies the number of parameters in the function. We will return to the details of this template in section 4. Now it suffices to say that the elementary model holds the parameters of the mathematical function to be calculated as *automatically differentiable numbers* in the proxy data member.

Fig. 4 shows a complete class definition of an example class derived from elementary_model. These derived classes contain the actual mathematical formulae of the model function (the eval function). In addition, only two simple utility functions (create and get_class_name) are needed. These are the only requirements for each elementary function class and it is thus very easy to extend the function hierarchy to cover new function types.

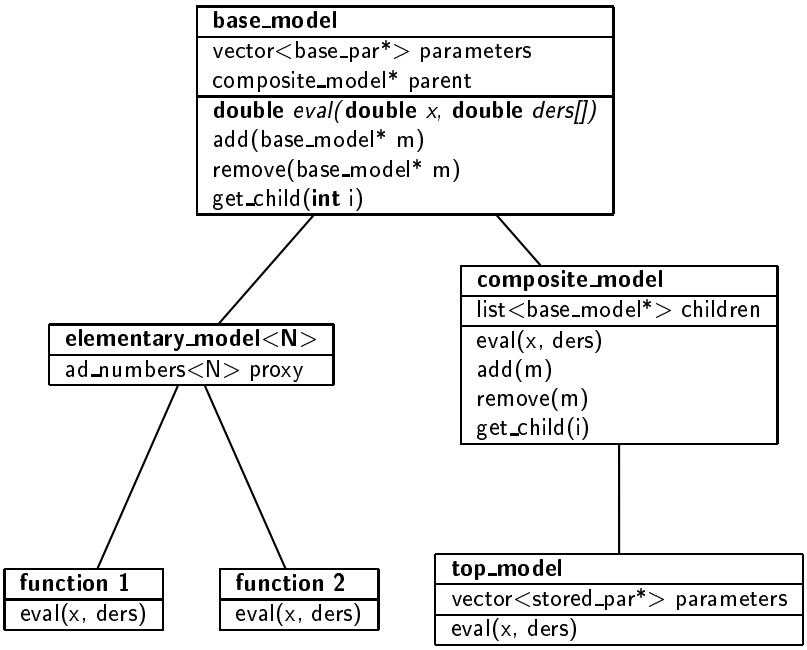


Figure 3: Model function class hierarchy.

Gamma et al. [8] have proposed some general methods for representing hierarchical structures in an object-oriented language. This model function hierarchy can be seen as a version of the *Composite design pattern*. Regarding the implementation issues of this pattern discussed by Gamma et al. we have chosen to maintain explicit parent references implemented as a pointer in the base_model class. We also chose to maximise the interface of the base_model. This means that, e.g., operations for manipulating the list of children of the composite models (add, remove) are declared and defined in base_model. This gives transparency for the client but on the other hand the operations do not have a meaning for elementary models. Therefore, by default, the operations add and remove fail (e.g. by raising an exception) and the functions are overridden in the composite_model class to give them meaningful definitions.

Not all functions are shown in the class diagram of Fig. 3. The base_model class also defines functions for adding and removing parameters as well as functions for naming the models. The *virtual constructor* [8, 1] mechanism is utilised in the object construction, requiring the two virtual functions, create and get_class_name, to be overridden in each derived class.

```
class lorenz : public elementary_model<4> {
public:
  lorenz* create() {return new lorenz(); }
  string get_class_name() {return "lorenz"; }
  enum {amp, freq, damp, ph };
  double eval(double x, vector<double>& ders) {
    return store_derivatives( ders,
      par(amp)*cos(2*pi*par(freq)*x + par(ph)) * exp(-x*par(damp))); }
};
```

Figure 4: Definition of an example function derived from the elementary_model class.

3.2 The parameter hierarchy

The parameter of a model function is basically just a value of some floating point type. However, the same parameter value may be shared by several component functions or there may be other dependencies between parameters. Hence, not all parameters of the component functions store a value. As a consequence, just representing a parameter as a floating point number is not sufficient to allow the component models to use the parameters in a uniform way. Therefore parameters are represented as classes from the parameter hierarchy (Fig. 5).

The base_par class is the topmost class of the hierachy and provides the common interface, the functions get_value and get_derivative for retrieving the value and initial derivative of the parameter. The stored_par class represents actually stored, adjustable parameters. The dependent_par class is the base class for dependent parameters and linear_par is for expressing linear relations between parameters. Other dependencies may be implemented by deriving new classes from the dependent_par class.

Each dependent parameter holds a pointer to another parameter, a *parent* parameter. The value is resolved by asking the value of the parent recursively until finally an instance of a stored_par class will end the recursion. The same mechanism applies for derivatives. The get_derivative function evaluates the derivative with respect to the underlying stored parameter. For stored_par this is 1 (the derivative of a variable with respect to itself is 1), while for linear parameters we get it by multiplying the derivative of the parent with the linear factor (see the code outlined in Fig. 5).

All parameters also maintain a child list and a pointer to the model function owning the parameter. The dependent parameters contain a vector of *parameter modifiers* such as the coefficients of the linear relation. The number of these modifiers is fixed for each derived class and given in the constructor.

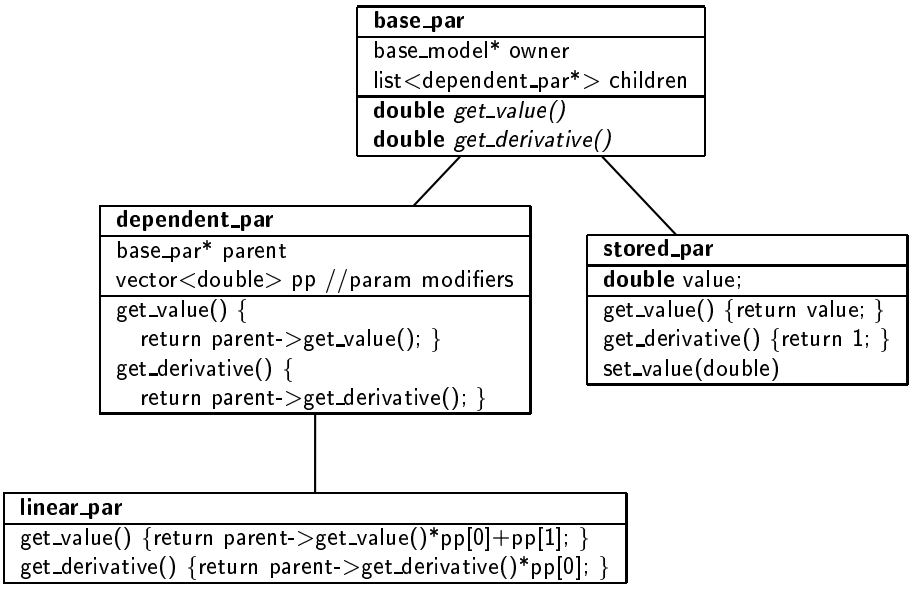


Figure 5: Parameter hierarchy.

3.3 Enforcing the consistency

The data structure for representing structural functions consists of several objects from the function and parameter hierarchies (Fig. 6). It is a combination of two object trees, both maintaining child node lists and parent pointers. In addition, the nodes of the model tree may own nodes of the parameter tree. This relation is represented as a list in the model tree node and a corresponding owner pointer in the parameter tree node. Furthermore, a vector of references to the adjustable parameters in the parameter hierarchy is maintained in the topmost model function.

To be able to guarantee the consistency of such a complex structure the construction and manipulation of the objects involved in the data structure must be controlled tightly. Though not shown in the class definitions, the creation and destruction of models is not part of the public interface of the classes, instead the creation of the objects is delegated to a special *creator* object and the destruction is performed from within the member functions of the classes of the hierarchy.

The final data structure maintains several invariances. The child list of a composite model is kept consistent with the parent references of the children. The same applies to child/parent relation in the parameter hierarchy as well as the parameter/owner relation between model functions and parameters.

The relation between parameters and models is further restricted. A parameter and its descendant can not be owned by different function hierarchies. Furthermore, the owner function of a dependent parameter must be a descendant of the owner of the parent parameter. Some of the invariances are guaranteed automatically by

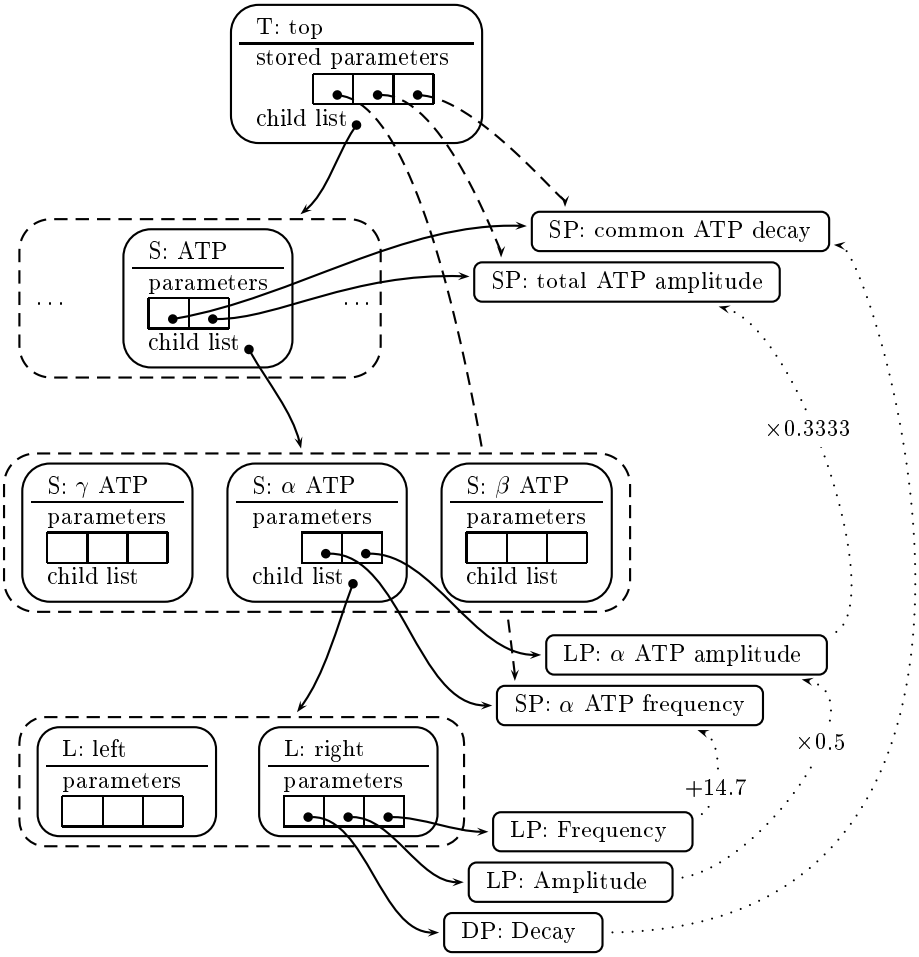


Figure 6: Instantiated objects and their relations illustrated in the NMR case (part of the ATP molecule). Solid lines represent ownership relation, while dashed lines are non-owning pointers. Dotted lines are parent links. The class of each object is given in parenthesis (C=composite, L=lorenz, LP=linear parameter, DP=dependent parameter, SP=stored parameter). Along the parent links of the parameters are the formulas for computing the values of linear parameters.

the restricted object construction. Others are enforced by raising an exception if a user tries to perform an operation which conflicts with an invariance condition.

3.4 Constructing model functions

The starting point of a model function is an instance of the top_model class. After it has been created, component models can be added to its child list.

The construction of objects is delegated to a special creator object implementing the virtual construction mechanism. The purpose of this is to make the client code, which initiates the creation of objects, independent of the changes in the elementary function classes.

The class of the object to be created is specified as a class name string at runtime. This is a convenient way of initiating object construction. Since the object creation task is most likely initiated by a user command, it is quite natural to specify the class as a class name string. The user may, e.g., have selected the class from a selection list.

The creation mechanism requires each class to *register* itself (one line of code) and define the virtual functions get_class_name and create. Otherwise the creation mechanism is totally independent of the derived classes: e.g., adding new elementary functions to the model hierarchy does not have any effect on the client code. For details of the virtual construction mechanism, see [8] describing several creational design patterns.

3.5 Model editor

As an example of a user interface for specifying structural model functions, Fig. 7 shows a snapshot of the model editor we have written. The structural function tree is visible on the left and the parameters of the currently selected model on the right. The names of the functions as well as the parameter names and values can be edited freely on the spot. There are buttons and menu commands for adding and removing functions and parameters, defining relations between parameters and storing and retrieving models. As pointed out above, the code of the model editor is totally independent of the particular elementary function classes derived from the model function hierarchy.

3.6 Persistence of model objects

In addition to methods for creating and modifying the structural model functions, means for their storage and retrieval are needed. In object-oriented systems, the ability of objects to live beyond the lifetime of the program is called persistence. It can be achieved using *object serialisation*, the common approach used in commercial class libraries such as MFC [12] and OWL [13]. This approach relies on virtual construction mechanism and requires the programmer to specify reading and writing methods for each persistent class.

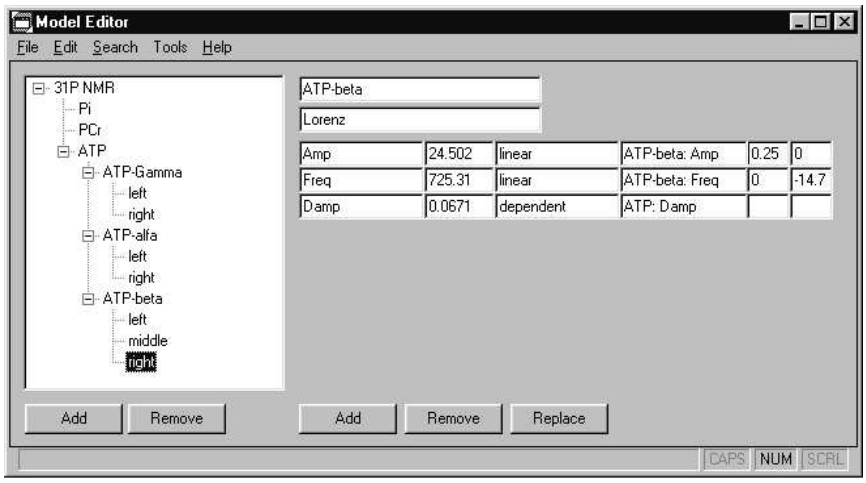


Figure 7: Snapshot of our model editor.

Virtual construction is already included in the model and parameter hierarchies. Furthermore, when the model hierarchy is extended, no new data members need to be introduced in the derived elementary function classes. Therefore the read and write functions can be inherited and need not be specified. Consequently, the implementation of persistence can be encapsulated entirely into the core classes of the model and no changes are required when new classes are added to the hierarchies.

4 Evaluation of function values and derivatives

In this section, the function value and derivative computation in our model is explained. The concept of automatic differentiation is described and it is shown how to calculate the derivatives of structural functions easily and yet effectively with this technique.

4.1 Automatic differentiation

The derivatives are traditionally calculated either symbolically or by using divided differences. The former may be quite difficult and error-prone while the latter introduces truncation errors and may be inaccurate and inefficient. Automatic differentiation provides an appealing alternative.

In automatic differentiation, the derivatives are computed by the well-known chain rule, but instead of propagating symbolic functions, numerical values are propagated along the computation. The evaluation of the function and its derivatives are calculated simultaneously using the same expressions. There are several descriptions about automatic differentiation [15, 1, 3] and also software packages

available [9, 4]. Some packages preprocess the source code to add the necessary statements for computing the derivatives. Other packages, using programming languages that support operator overloading, implement the differentiation as a class library without the need for a separate precompilation.

There are interesting computational issues concerning the implementation of automatic differentiation. The chain rule can be used either in *forward* or *backward* mode or in something between. The implementation involves a tradeoff between time and space complexity. In this article the forward mode automatic differentiation is used. It is simple and fits very well in this particular application as will become clear below.

In forward mode automatic differentiation, instead of computing with scalar values, we compute with automatically differentiable numbers (ADN) $\langle \mathbf{f}, \nabla \mathbf{f} \rangle$. An ADN consists of a value and a vector of partial derivatives of a function at a given point. When building expressions with these objects, at the leaf level of the expression tree $\mathbf{f}$ is either a variable or a constant. When differentiating with respect to N variables, the derivative of the ith variable is represented as the ith canonical unit vector of length N and the derivative of a constant with a zero vector. For example, when differentiating with respect to three variables x, y, z the constant 3.14 is expressed as $\langle 3.14, (0,0,0) \rangle$ and the variable y as $\langle y, (0,1,0) \rangle$. Computation with these objects utilises the chain rule of derivatives.

$$\left.\frac{\partial}{\partial t} f(g(t))\right|_{t=t_0} = \left(\left.\frac{\partial}{\partial s} f(s)\right|_{s=g(t_0)} \right) \left(\left.\frac{\partial}{\partial t} g(t)\right|_{t=t_0} \right) \tag{4}$$

As an example, consider the two-derivative case for function $y + \sin(x^2)$. Starting with $\langle y, (0,1) \rangle + \sin\langle x, (1,0) \rangle^2)$ by squaring x, we get $\langle y, (0,1) \rangle + \sin(\langle x^2, (2x, 0) \rangle)$. Taking the sine gives $\langle y, (0,1) \rangle + \langle \sin(x^2), (2x\cos(x^2), 0) \rangle$ and finally the addition with y gives $\langle y + \sin(x^2), (2x \cos x^2), 1) \rangle$. For numerical work, the computation is not done symbolically, rather the actual values of the function and its derivatives are calculated and propagated through the expression. Given $x = 2, y = 4$ the same example becomes

$$\begin{aligned} \langle 4, (0,1) \rangle + \sin(\langle 2, (1,0) \rangle^2) &= \langle 4, (0,1) \rangle + \sin(\langle 4, (4,0) \rangle) = \\ \langle 4, (0,1) \rangle + \langle 0.06976, (1.9951, 0) \rangle &= \langle 4.06976, (1.9951, 1) \rangle. \end{aligned}$$

The method can be applied to any machine-computable function. All that is needed is to code the differentiation rules for simple functions and operations. Then any function composed of those elementary functions can be differentiated automatically. In C++ this means overloading common functions and operators for objects described above.

The forward mode automatic differentiation for calculating gradients can be computationally unattractive if applied blindly. If the gradient has n elements, the computation may require up to order of n as much time as computating the value of the same expression. However, in the case of partially separable functions the forward mode can be applied efficiently. If we consider the model function as a

whole, it may have quite a number of parameters, but the number of parameters of the individual elementary functions is typically rather low and known beforehand. Furthermore, different elementary functions are only related via a summation expression, which means that also the derivatives are just summed together. Consequently, we use automatic differentiation in computing the local gradients of the elementary functions and update the calculated values via pointers to the common derivative vector.

The computing time of the local gradients can be further reduced. By using C++ templates, moderate size derivative vectors of ADNs can be replaced with special sparse vectors to yield very efficient code [10]. This method was utilised in the test runs described in section 4.4.

4.2 The function evaluation process

The model function is evaluated by calling the eval function of the topmost class, which will traverse all the contained models and calculate their cumulative values at a given point. The derivatives are computed simultaneously using automatic differentiation. The derivative vector is passed as a parameter to the eval function. First the resulting derivative vector or gradient is initialised to zero. Each elementary function reads the values and initial derivatives of the parameters (with the get_value and get_derivative functions) and constructs ADNs from them. If the elementary function has n parameters, ADNs having n-dimensional derivative vectors are used. The mathematical expression is then evaluated using ADNs and each elementary function updates the resulting derivative values to the actual gradient vector. This is accomplished with a call to store_derivatives function defined in the elementary_model template (see Fig. 4), which adds the derivative values to the right positions of the gradient.

After the whole function tree has been traversed, the function value is returned and the gradient is available as the derivative vector passed to the eval function.

4.3 Computational efficiency

With regard to the computational efficiency the evaluation strategy includes a few pitfalls. Firstly, dynamic binding is applied in the eval function invocations. There is an inherent additional cost in a call to a dynamically bound virtual function compared with a statically bound function [7]. Furthermore dynamic binding precludes the use of inlined functions. Inline expansion can speed up function calls and is beneficial for small functions. However, in this case the computational cost of the function call is probably minor compared to the cost arising from the evaluation of the actual mathematical formulae of the elementary functions, recalling that the derivatives are also calculated in the same function. Considering this, the relative cost of the slightly slower function call is most likely insignificant in this case.

Secondly, dynamic binding is also applied between parameters in the get_value and get_derivative functions. In this case the extra cost may be notable. The evaluation of an elementary function having n parameters would yield at least n

virtual function calls to fetch the parameter values. The number of calls is larger if dependent parameters are involved. However, in model fitting tasks the model function is evaluated repeatedly at several points, without changing the parameter values. Taken the example from NMR spectroscopy, the region of interest may contain thousands of points. Therefore the parameter values can be cached and only when the parameter values are changed, each elementary function reads the values and derivatives with the virtual get_value and get_derivative functions and stores the values to local proxy variables (ADNs). With this approach the relative cost of retrieving the parameter values via virtual functions is of little consequence. The caching is made transparent to the client code by maintaining a flag in the topmost class indicating whether the values in the proxy variables are valid or not.

Also, the updating of the local gradients to the global derivative vector must be efficient. This is implemented in the elementary_model template by maintaining a mapping from each local parameter index to an index in the global derivative vector. These mappings can be constructed prior to the first model evaluation. In this task the function tree must be traversed once, but this causes no efficiency problems, since the indices only change if the model function changes, i.e., new component functions are added or removed. At evaluation time the only additional cost is an extra indirection for each parameter.

Some cost may also arise if the composite models in the function tree contain many levels (e.g. in the ATP compound). From the computational point of view, it is not necessary to traverse all composite functions during the evaluation, rather it is sufficient to call the evaluation functions of the elementary models directly and save the cost of a few virtual calls. This is easy to implement by maintaining a separate list of the leaf nodes in the top_model class, which we did in the test runs.

4.4 Test runs

To assess the efficiency of the model some test runs were performed. As a test case, we used formulae from the NMR case consisting of 10 component functions having 24 adjustable parameters altogether. Five different alternatives to perform the function and derivative computations were programmed:

1. A tailored low-level C-code with analytical derivatives.
2. The presented OO model with analytical derivatives.
3. The OO model with automatic differentiation.
4. A straightforward OO implementation, without any caching.
5. A low-level implementation of the function with finite difference value approximations of the derivatives.

In the tailored low-level implementation, the model function was totally fixed at design time, so any change in the function requires changes in the code. The code was hand-optimised to a reasonable level (not making any processor specific tricks).

Implementation	Relative time
1. Tailored low level C-code	1.00
2. OO model with analytical derivatives	1.07
3. OO model with automatic differentiation	1.29
4. Straightforward OO implementation	2.48
5. Divided difference approximations	16.52

Table 1: Relative evaluation times of the different methods for computing the value and derivatives of the NMR model.

All subexpressions were calculated only once and all relations between parameters were directly written into the code as effectively as possible. It is fair to say that the code used was as fast as possible.

In the second case, the OO model presented was used, but the derivatives of the elementary functions were calculated analytically. This case should roughly represent the extra cost originating from the dynamic binding of the model functions, as well as the cost arising from not coding the dependencies between the parameters directly.

In the third case, the OO model was used with derivatives computed using automatic differentiation. Table 1 shows the results and confirms the extra cost being quite acceptable compared with the flexibility the model offers.

In the fourth case, no proxies for parameters were used, rather the initial values and derivatives were retrieved during each evaluation using the virtual function invocations. This demonstrates that the performance may drop significantly if the programmer is not aware of the principles affecting efficiency in OO programs.

In the fifth case, derivatives were approximated with divided difference values. The benefit of this alternative is that only the code for evaluating the value of the model function is needed. The performance is, however, very poor requiring n evaluations of the model function, where n is the number of elements in the gradient. Also, the accuracy is harder to assess.

The test runs were performed under Linux on Intel Pentium processor. The C++ compiler used was KAI C++ 3.2.d with optimisation flags +K2 -O3.

5 Conclusions

An object-oriented model for parameter estimation of partially separable function was described. The model achieves two goals. Firstly it gives an easily extendible OO framework for representing partially separable functions in a structured way, resembling the physical real-life interpretation and mathematical structure of the functions. Secondly, it offers an interface to an optimisation algorithm, namely a vector of adjustable parameters and a function capable of computing the value and derivatives of the model function efficiently.

To achieve the first goal, the model separates the commonalities of partially

separable functions from the specific mathematical formulae. The formulae are encapsulated to a few very simple classes. It is therefore easy to apply the model to different problem domains, since changing these classes or adding new ones to the model does not affect the client code using the model. Furthermore, relations between parameters are handled by the model and they do not complicate the mathematical expressions of the component functions.

The derivatives needed in the parameter estimation are obtained using automatic differentiation. Hence, there is no need to hand-code analytical derivatives or use divided difference values.

Considering the second goal, the calculation of function values and derivatives is efficient. In our example case from NMR spectroscopy, the evaluation of the OO model required only 29% more time than a low-level tailored implementation of the same function. As a compensation, in the OO model the final function as well as relations between parameters can be specified at run-time, the model is easily extendible to cover new component functions and no hand-coded derivatives are required.

To sum up, the paper gives practical guidelines for implementing an efficient OO computational kernel for partially separable functions. With an example, we showed that OO programming offers substantial benefits, such as higher abstraction level, code reuse, flexibility and handling of complexity for numerical programming as well. Furthermore, the advantages can be achieved with a moderate loss of performance.

References

[1] Barton J. J., Nackman L. R.: Scientific and Engineering C++, Addison-Wesley, Reading Massachusetts 1994.

[2] Bazaraa M.S., Sherali H.D., Shetty C. M.: Nonlinear Programming: Theory and Algorithms, 2nd Edition, Wiley 1993.

[3] Editors: Berz M., Bischof C. H., Corliss G. F., and Griewank A.: Computational Differentiation - Techniques, Applications, and Tools, SIAM, Philadelphia Pennsylvania 1996.

[4] Bischof C. H., Carle A., Corliss G. F., Griewank A., Hovland P.: ADIFOR: Generating derivative codes from Fortran programs, Scientific Programming, **1** (1992) 1-29.

[5] Bovée W. M. M. J.: Quantification in in vivo NMR, Spectral editing, in: Magnetic Resonance Spectroscopy in Biology and Medicine, eds. de Certaines J. D., Bovée W. M. M. J., Podo F., 181-207, Pergamon, Oxford 1992.

[6] Derome A. E.: Modern NMR Techniques for Chemistry Research, 63-90, Pergamon, Oxford 1991.

[7] Driesen K., Hölzle U.: The Direct Cost of Virtual Function Calls in C++, ACM Sigplan Notices, OOPSLA'96 Proceedings, **31** (1996) 306-323.

[8] Gamma E., Helm R., Johnson R., Vlissides J: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading Massachusetts 1995.

[9] Griewank A., Juedes D., Utke J.: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++, ACM Transactions on Mathematical Software, **22** (1996) no.2, 31-167.

[10] Järvi J.: A PC program for automatic analysis of NMR spectrum series, Computer Methods and Programs in Biomedicine **52** (1997) 213-222.

[11] Majoras R. E., Richardson W. M., Seymour R. S.: An object-oriented approach to evaluating multiple spectral models, Journal of Radioanalytical and Nuclear Chemistry **193** (1995) 207-210.

[12] Microsoft, Microsoft Foundation Class Library, Microsoft Corporation.

[13] Borland, Borland C++ 5 Programmer's guide, Borland International, 1996.

[14] Press W.H., Teukolsky S.A., Vetterling W.T., Flannery B.P.: Numerical Recipes in C: the Art of Scientific Computing, 2nd Edition, Cambridge University Press, New York 1992.

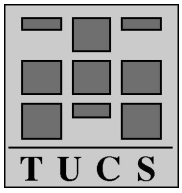
[15] Rall L.B.: Automatic differentiation: Techniques and Applications, Lecture Notes in Computer Science 120, Springer-Verlag, Berlin 1981.

[16] Robinson A.D.: C++ Gets Faster for Scientific Computing, Computers in Physics **10** (1996) 458-462.

[17] van Tongeren B. P. O., Boxman R. D. C., Deumens J. W., van Leeuwen J. P., Mehlkopf A. F., van Ormondt D., de Beer R.: QUANSIS, An object-oriented data-analysis system for in vivo NMR signals, Journal of Magnetic Resonance Analysis, **2** (1996) 75-84.

Tuples and multiple return values in C++

Jaakko Järvi



Turku Centre for Computer Science
TUCS Technical Report No 249
March 1999
ISBN 952-12-0401-X
ISSN 1239-1891

Abstract

A generic *tuple* class, capable of storing an arbitrary number of elements each being of arbitrary type, is presented. The class offers a concise means to return multiple values from a function. Instead of using refrerence parameters to pass data out of functions, the function return type is an instantiation of the tuple template, grouping the values to be returned. The semantics of the tuple class is analogous to the pair template in the C++ standard library, thus the usage of tuples is convenient and intuitive. The implementation is described in details.

Efficiency considerations are addressed. It is shown that with modern C++ compilers, there is no extra runtime overhead caused by using tuples. However, due to excessive template instantiatons, there is some effect on compilation times and memory usage during compilation. This effect is unlikely to be significant in real programs.

Keywords: generic programming

TUCS Research Group
Algorithmics group

1 Introduction

In most common programming languages (including C++) functions can return only a single value. This is a reasonable restriction, although quite often there is a need to pass several related values from a function to the caller. For example, consider the numerous cases, where in addition to the actual result, one or more status codes are given. Particularly typical multiple return values are in numerical libraries, where besides the extensive need for status codes, the actual results themselves often have several components. E.g., a common matrix decomposition operation, singular value decomposition (SVD), takes a matrix and decomposes it to two matrices and a vector. Still another example is the usage of the `pair` template with STL maps.

To circumvent the restriction of a single return value, reference or pointer parameters can be used to pass data out of functions. Alternatively a struct or class type can be defined to group the values to be returned into a single object. In the first approach, the distinction between input and output parameters is not always obvious and one may need to add extra comment lines just for clarifying this (although disciplined usage of common idioms, such as using non-const pointers and references only for output parameters does help). The second approach is conceptually better: there are no multiple return values, just a single value, which happens to be a composition of several related values. All the parameters are input parameters and the return value is the sole output.

Consider the SVD example. One would either write

```
void SVD(const Matrix& M, Matrix& U, Vector& S, Matrix& V);
```

and leave it to the callers responsibility to provide `U`, `S` and `V` suitably initialized, or use the latter approach:

```
struct SVD_result {
  Matrix U, V;
  Vector S;
  // possibly a constructor etc.
};
SVD_result SVD(const Matrix& M);
```

Even though in this case the distinction between input and output values is clear, it is a burden to write small classes just to serve as a collection of return values of some individual function. New unnecessary names are added to the namespace, and in addition to the name and prototype of a function, the caller must also know the return class and its behavior. In this article,

tuples are proposed as an alternative way of returning multiple values from a function.

1.1 Tuples

Tuples are fixed size collections of objects of arbitrary types. They provide a concise means for multi-valued returns. The return type of a function can simply be defined as a tuple, e.g.:

```
<Matrix, Vector, Matrix> SVD(Matrix); // pseudo code
```

The intention of the declaration is instantly clear and there is no need to specify an artificial wrapper class for the result.

Some programming languages, such as Python[1], Scheme[2] or ML[3], have tuple constructs or analogous mechanisms for multiple return values. There is also an ongoing effort to include tuples to Eiffel[4]. C++ has no direct support for tuples. However, the generic features of the language offer potential solutions. The pair template in the standard library implements 2-tuples, allowing us to write, for example

```
pair<Matrix, Matrix> LU(Matrix); // another matrix decomposition
```

Although the pair suffices for 2-tuples, it does not provide a general solution. It is of course possible to write templates for triples, quadruples, and so on, but where to stop?

This article describes a solution, which covers all lengths of tuples with a single template. The solution is completely type-safe requiring no runtime checks. Furthermore, with modern optimizing C++ compilers, the usage of tuples (creation and element access) has no performance penalty compared with using normal structs as return values. Also, independent of the number of elements, *tuple* is the common name for all tuple types (instead of tuple2, tuple3, etc.). The template definitions set a certain predefined upper limit for the length of tuples, but in any case tuples of several dozens of elements are allowed, which should be enough for most of the imaginable uses.

Note that the code presented relies on the new template features of the C++ standard[5], such as member templates, partial specialization and explicit specialization of function templates. Not all compilers currently support these features. The code in this article was successfully compiled with the Egcs and KAI C++ compilers, both being on the leading edge with respect to the conformance to the new standard.

2 Tuple Template

A tuple template must be able to store an arbitrary number of elements of arbitrary types. A template having this capability is surprisingly simple:

```
struct nil {};

template<class HT, class TT >
struct cons { HT head; TT tail; };

template<class HT>
struct cons<HT,nil> { HT head; };
```

As the name suggests, the definition corresponds to the LISP dot pair. Instantiating `TT` with a `cons` recursively leads to a list structure (see [6] for a more detailed description of these *compile time recursive objects*). E.g., the instantiation representing the tuple type `<Matrix, Vector, Matrix>` is

```
cons<Matrix, cons<Vector, cons<Matrix, nil> > > aTuple;
```

This instantiation defines a nested structure of classes, containing the tuple elements as nested member variables. The `nil` class is just an empty placeholder class. The elements can be accessed with the usual syntax (e.g. `aTuple.tail.tail.head` refers to the third element). This `cons` template is the underlying construct for representing tuples.

Even though the `cons` template is sufficient for representing the structure of tuples, the syntax is not usable. Directly defining tuples as dot pair lists with the `cons` template would be rather awkward, and indeed a better syntax can be attained. The objective is to be able to write type declarations, such as `tuple<int>`, `tuple<float, double, A>` etc. Since the number of template parameters of a generic class can not vary and neither is the definition of several templates with the same name possible, the solution is to use default arguments for template parameters:

```
template <class T1,class T2=nil,class T3=nil,class T4=nil>
struct tuple ...
```

Now this definition allows between 1 to 4 template arguments. The unspecified arguments are of type `nil`. As pointed out above, we need to set an upper limit for the number of elements in any tuple. To make the code sections short, the limit of four elements is used here. It is, however, straightforward to extend the tuple to be able to handle more elements.

Now the `tuple` template provides the desired interface, whereas the `cons` template implements the underlying structure of tuples. The interface and

implementation can be connected via inheritance. A tuple inherits a suitable instantiation from the cons template: `tuple<float, int, A, nil>` for example inherits from `cons<float, cons<int, cons<A, nil> > >`. It is reasonable to assume that a tuple only stores its non-nil elements, hence `nil` classes are excluded from the `cons` instantiation. To be able to implement this inheritance relation, we need a means to express the type to be inherited using the template parameters of the tuple.

2.1 Mapping tuple types

A recursive *traits* (more on traits, see [5]) class is needed to define the mapping from tuple parameters to a suitable cons instantiation:

```
template <class T1, class T2, class T3, class T4>
struct tuple_to_cons {
  typedef
    cons<T1, typename tuple_to_cons<T2, T3, T4, nil>::U > U;
};

template <class T1>
struct tuple_to_cons<T1, nil, nil, nil> {
  typedef cons<T1, nil> U;
};
```

The sole purpose of this class is to define a mapping from the 4 template parameters of `tuple` to a corresponding instantiation of the `cons` template. This is given by the typedef `U` in the `tuple_to_cons` class. It defines a dot pair, where the head type is the first template parameter `T1` and the tail type is defined recursively. The recursive definition instantiates the `tuple_to_cons` template again, with `T1` dropped from and `nil` added to the parameter list. This is repeated, until the partial specialization of `tuple_to_cons` matches.

As an example, consider `tuple<float, int, A, nil>`. To resolve the underlying type of the typedef `tuple_to_cons<float, int, A, nil>::U`, the following chain of instantiations occurs:

```
tuple_to_cons<float, int, A, nil>::U
≡ cons<float, tuple_to_cons<int,A,nil,nil>::U >
≡ cons<float, cons<int, tuple_to_cons<A,nil,nil,nil>::U > >
≡ cons<float, cons<int, cons<A,nil> > >.
```

In the last step, the partial specialization is applied.

Using the `tuple_to_cons` type mapping the tuple class can now be defined as:

```
template <class T1,class T2=nil,class T3=nil,class T4=nil>
struct tuple : public tuple_to_cons<T1, T2, T3, T4>::U {...};
```

Now it is clear that tuple instantiations up to four elements with arbitrary types are valid and inherit a cons instantiations capable of storing the elements. For example, the following definitions are all correct instantations of the tuple template.

```
tuple<int, int>
tuple<Matrix, Vector, Matrix>
tuple<A, B, tuple<C, D>, E>
```

3 Construction semantics

The templates above define just the structure of tuple types. For tuples to be usable, we need convenient mechanisms for constructing tuples and accessing their individual elements. The construction and element access semantics can be made consistent of the semantics of the pair template in the standard library:

```
template< class T1, class T2>
struct pair {

  T1 first; T2 second;

  pair() : first(T1()) , second(T2()) {}
  pair(const T1& x, const T2& y) : first(x), second(y) {}

  template<class U, class V>
    pair(const pair<U, V>& p)
      : first(p.first), second(p.second) {}
};
```

Hence, by default a pair is initialized to the default values of its element types. The elements may also be given explicitly in the construction. The member template is a 'copy constructor', which allows type conversions between elements. This is the construction semantics a general tuple template should have as well. Furthermore, the element access should be as straightforward as with pairs (`p.first`, `p.second`, etc.). Let us first focus on the construction semantics.

3.1 Tuple constructor

The tuple template above allows up to four elements. It is therefore obvious that the tuple constructor should have four parameters. But tuples can also

be shorter. For an n-tuple, there should be a constructor taking n parameters. To avoid writing a separate constructor for each different length, default arguments can be used. The definition of the tuple template becomes:

```
template <class T1,class T2=nil,class T3=nil,class T4=nil>
struct tuple : public tuple_to_cons<T1, T2, T3, T4>::U {

  tuple( const T1& t1=wrap<T1>(), const T2& t2=wrap<T2>(),
         const T3& t3=wrap<T3>(), const T4& t4=wrap<T4>())
    : tuple_to_cons<T1, T2, T3, T4>::U(t1, t2, t3, t4) {}
};
```

The constructor has a distinct parameter for each element to be stored. The parameters are passed directly to the inherited cons template instantiation. We have not yet defined any constructors for the cons template, so just assume for now that the constructor works reasonably. Each parameter is given a default value. Though looking somewhat odd, the default arguments are expressions that merely create and return a new default object of the current element type. Given this definition, a constructor for an n-element tuple can be called with 0 to n parameters and the elements left unspecified are constructed using their default constructors. Especially, the unused `nil` objects are created by the default argument expressions, hiding their existence entirely from the user of the tuple template.

3.1.1 Default arguments

Why not just use `T()` instead of `wrap<T>()` as the default argument? Suppose `A` is a class with no public default constructor. Then the constructor of the `tuple<A>` instantiation is `tuple<A>(const A& t1 = A(), ...)`. This results in a compile time error, since the constructor call `A()` is invalid. Hence a type without a default constructor would not be allowed as an element type of a tuple. This is the case even if the default argument is never used. The solution is to wrap the default constructor call to a function template[1]:

```
template <class T> inline const T& wrap() { return T(); }
```

Now the constructor of `tuple<A>` becomes:

```
tuple<A>(const A& t1 = wrap<A>(), ...)
```

[1] In the egcs version 1.0.2. this is not yet supported. A class template with a static function: `template<class T> struct wrap { static T f() { return T(); };` must be used instead of a function template. The call becomes: `wrap<T>::f()`.

If the first parameter is always supplied, the `wrap<A>` template is never instantiated.[2] The compiler only checks the semantic constrains of the default argument and finds out that `wrap<A>()` is indeed a valid expression. Only if the default argument is really used, the compiler is allowed to instantiate the body of the wrap function template (and flag the error if there is no default constructor for the type in question)[5, section 14.7.1.].

3.2 Constructing dot pairs

The tuple constructor passes all four parameters to the constructor of the inherited `cons` instantiation. Hence, the `cons` constructor also has four parameters. This constructor is a member template, where only the type of the first parameter is fixed and the remaining parameter types are deduced. Here are the definitions of the cons templates:

```
template<class HT, class TT> struct cons {
  HT head; TT tail;

  template <T2, T3, T4>
    cons(const HT& t1, const T2& t2,
         const T3& t3, const T4& t4)
      : head(t1), tail(t2, t3, t4, nil()) {}
};

template <class HT> struct cons<HT, nil> {
  HT head;
  cons(const HT& t1, const nil&, const nil&, const nil&)
    : head (t1) {}
};
```

The constructor initializes the head with the first parameter and passes the remaining parameters to the tail's constructor recursively, until all but the first parameters are nil. At each level, one element is initialized. Note that even though only the first parameter's type is directly bound, the types of the remaining parameters are in fact also mandated by the instantiation of the `cons` template. Hence, a constructor call with erroneous argument types result in a compile time error at some point during the recursive instantiations.

As an example, let us consider in detail a particular constructor call `tuple<Matrix, Vector, Matrix>(U,S,V)`. When the call is encountered,

[2] Version 3.3.c of KAI C++ incorrtecly instantiates the default argument. According to their support, this is to be addressed in their next major release. For a workaround, contact the author.

the tuple template is instantiated: the missing fourth template parameter is first added and `tuple<Matrix, Vector, Matrix, nil>` becomes the complete instantiated type. The `tuple_to_cons` traits class computes the list type `cons<Matrix, cons<Vector, cons<Matrix, nil> > >` to be inherited. The tuple constructor is called with arguments `U`, `S` and `V`. The fourth parameter is not given, so the default argument is applied: `wrap<nil>()` is used to construct an empty object of type `nil`. Now the cons constructor is called with arguments `U`, `S`, `V` and `nil()`, which recursively initializes the elements of the tuple with `U`, `S` and `V`.

3.3 Constructor allowing element-wise conversions

The constructors analogous to the third constructor of the pair template are still to be defined. The intention is to allow 'copy' construction from another tuple with different element types, if the elements can be implicitly converted. For example, `tuple<int, double, int>` could be initialized with an object of type `tuple<char, int, int>`. For pairs, this is useful mostly in situations, where the pairs are created using the `make_pair` function template [8]. There does not seem to be a generic way to implement a corresponding `make_tuple` function (other than explicitly writing a function for each tuple length). It is thus a matter of taste, whether a 'converting' copy constructor is needed for tuples, but nevertheless it is not difficult to implement. An additional member template constructor is required in the tuple template:

```
template <class T1, class T2, class T3, class T4>
  template<class U1, class U2>
  tuple<T1,T2,T3,T4>::tuple(const cons<U1, U2>& p)
    : base(p) {}
```

In both of the cons templates (primary and specialization), a new constructor is needed.

```
//primary template
template<class HT, class TT>
  template<class HT2, class TT2>
  cons<HT,TT>::cons(const cons<HT2, TT2>& u)
    : head(u.head), tail(u.tail) {}

//specialization
template<class HT>
  template<class HT2>
  cons<HT, nil>::cons(const cons<HT2, nil>& u)
    : head(u.head) {}
```

The tuple constructor merely delegates the copy to the base class. In the base class, the copy constructor of each member is called along the recursion. Hence, the converting copy is allowed, if the types are element-wise compatible. Otherwise a compile time error results.

4 Accessing tuple elements

With the above definitions, the element access is tedious. For example, one must write `aTuple.tail.tail.tail.tail.head` to refer to the fifth element of a tuple. However, as stated above, element access should be as convenient as with pairs. Since the elements of tuples have no names (such as first, second, etc.), numbers are used to refer to them. With the aid of some additional template definitions, the `N`th element can be referred with the expression `get<N>(aTuple)`.[3]

Here `get` is a function template, where the index of the element to be accessed is given as an explicitly specified integral template argument. Obviously, the access mechanism must be recursive. However, the `get` function template can not directly be defined recursive. It is not possible to define a specialization with respect to a template parameter, which must be explicitly specified. Therefore, the recursion is implemented as a static member template of the class `nth`:

```
template<int N> struct nth {
  template<class HT, class TT>
  static void* get(cons<HT, TT>& t) {
    return nth<N-1>::get(t.tail);
  }
};

template<> struct nth<1> {
  template<class HT, class TT>
  static void* get(cons<HT, TT>& t) { return &t.head; }
};
```

Now `N` is a template parameter of a class and the specialization for `N`=1is thus allowed.

Since tuples are inherited from some `cons` instantiation, the element access functions can be defined to operate on `cons` dot pairs. The invocation

[3]The access functions can be defined as member templates as well, but since explicit specialisation is used, the `get` function must be explicitly qualified as a template. This would lead to the awkward syntax `aTuple.template get<N>()` instead of `aTuple.get<N>()`.

`nth<N>::get(aTuple)` returns a pointer to the `N`th element of the object `aTuple`. The access function works recursively returning the result of getting the (`N`-1)th element of the tail of `aTuple`. The specialization for `N`=1 merely returns the address of the head of the current dot pair. Note, that the access mechanism is safe with respect to the index parameter `N`, since an illegal index leads to a compile time error (no matching templates exist). Now we have a mechanism for getting the address of a given element, but not yet the type.

The type can be defined recursively: The type of the nth element of a tuple a equals the type of the $(n-1)$th element of the tail of a. With this definition in mind, we can write the corresponding recursive traits classes:

```
template <int N, class T> struct nth_type;

template <int N, class HT, class TT>
struct nth_type<N, cons<HT, TT> > {
  typedef typename nth_type<N-1,TT>::U U;
};

template<class HT, class TT>
struct nth_type<1, cons<HT, TT> > { typedef HT U; };
```

Now the type of the `N`th element of a dot pair type `T` can be written as `nth_type<N, T>::U`. Using this type definition, the actual `get` function template can be written as:

```
template<int N, class HT, class TT>
nth_type<N, cons<HT, TT> >::U&
get(cons<HT, TT>& c) {
   typedef
     typename nth_type<N, cons<HT, TT> >::U return_type;
   return *static_cast<return_type*>(nth<N>::get(c));
}
```

The `get<N>` function calls the `nth<N>::get` function to get the address of the `N`th element as a void pointer, casts it to the correct type and returns the result. Note that although the type information is seemingly lost, the cast from the void pointer is, however, entirely safe.[4]

As an example of the element access, consider accessing the third element of the tuple `a` defined as `tuple<Matrix, Vector, Matrix> a(U,S,V);` The invocation `get<3>(a)` triggers the following chain of instantiations:

[4]It is possible to carry the type information along in the `nth<N>::get` functions as well, but dropping it alleviates the task of the compiler considerably.

```
get<3,Matrix, cons<Vector, cons<Matrix, nil> > >(a)
→ nth<3>::get< Matrix, cons<Vector, cons<Matrix, nil> > >(a)
→ nth<2>::get<Vector, cons<Matrix, nil> >(a.tail)
→ nth<1>::get<Matrix, nil>(a.tail.tail).
```

Now the specialization `nth<1>` is used, and the head of `a.tail.tail`, which is the matrix `V`, is returned.

5 Efficiency

The tuple is intended to be an elementary utility template with widespread usage. Hence, efficiency is of utmost importance. However, the template definitions do not seem to be very efficient: The construction of tuples and accessing their elements requires several nested function calls. To access the `N`th element of a tuple with the `get<N>(aTuple)` function, `N`+1 functions altogether are invoked. However, the functions are all inlined 'one-liners'. In an optimizing C++ compiler, inline expansion eliminates the overhead of these functions and the address of the `N`th element is resolved at compile time.

The construction is analogous in this respect. Even though the construction of an n-tuple constructs n nested dot pairs, each constructor only effectively constructs its head and passes rest of the parameters forward. As the inline expansion is performed, the result is just the code performing the construction of the individual elements in a tuple. Particularly, the construction and destruction of the temporary `nil` objects are optimized away. This behavior was validated by experiments with the egcs (release-1.0.2, optimization flag -O) and KAI C++ (version 3.3c, flags +K3 -O2) compilers.

The obvious alternative for using the tuple template, is to explicitly write a struct containing an equivalent member variable for each tuple element. Suppose that a four-element tuple, composed of elements of types `A1`, `A2`, `A3` and `A4`, is needed. The alternative for `tuple<A1, A2, A3, A4>` is:

```
struct A {
  A1 first; A2 second; A3 third; A4 fourth;
  A(const A1& a1=A1(), const A2& a2=A2(),
     const A3& a3=A3(), const A4& a4=A4() )
      : first(a1), second(a2), third(a3), fourth(a4) {}
};
```

The comparisons between programs using these *explicitly written structs* vs. programs using tuples confirmed the anticipated behavior with both compilers. The compiled codes of object construction (e.g. the calls `A()` and

`tuple<A1, A2, A3, A4>()`) were essentially identical for both approaches, consisting only of the code arising from the construction of the individual elements. Similarly, both compilers were capable of computing the relative address of a given element in a tuple at compile time, thus eliminating all overhead arising from the element access. E.g. in the example above, `get<3>(aTuple)` yields no code, just a reference to the third element of the tuple.

To assess the results, several tests were performed, varying the length and element types of tuples. Egcs eliminated all overhead in all cases. Even a tuple of 256 elements was compiled: the compiler eliminated the 257 nested function calls of the invocation `get<256>(aTuple)` and resolved the address of the 256th element of the tuple at compile time!

The optimization capabilities of the KAI C++ depended on whether exceptions were used or not (the compiler has a flag for turning exceptions on and off). With exceptions turned on, the compiler only reached the zero overhead for relatively short tuples. With exceptions off, there were no difficulties in optimizing longer tuples as well.

5.1 Effect on compilation time

Due to excessive template instantiations, it is inherently slower to compile tuples than corresponding explicitly written structs. Further, the compilation requires a greater amount of memory. The compilation speed difference was measured with some simple tests. All template definitions were included as a header file (this is natural, since every function is inlined). Four pairs of test programs were generated. Each pair had a program using tuples and another equivalent program using explicitly written classes. The intention was to measure the direct compilation speed difference between tuples and explicit classes, so the tuple programs consisted of nothing else than tuple definitions, element access and functions (basically empty) that returned tuples. The results are shown in Fig. 1.

Another test was performed to evaluate the effect of tuple usage to compilation speed in a realistic program. An existing program was modified and an'average' C++ program was generated. It consisted of 120 functions. 25% of the functions had tuples as return values. The lengths of the tuples varied from 3 to 8. The program included a few standard headers (iostream and some STL headers) and used STL containers and algorithms to some extent. The compilation of this program was now less than 4% slower than the compilation of an equivalent program, which used explicitly written structs instead of tuples. The increase in the amount of memory needed in the compilation was between 5-10%. Consequently, the use of tuples increase

Tuple length	Egcs (T_t/T_s)	KAI C++ (T_t/T_s)
3	7.70	5.33
5	8.06	5.90
10	10.8	7.40
32	13.4	16.8

Figure 1: The relative compilation time of programs using tuples to equivalent programs using explicitly written structs (T_t = compilation time of tuple implementation, T_s = compilation time of explicitly written struct implementation). The times are not comparable across compilers.

compilation times and memory consumption, but the increases are likely to be insignificant on real programs.

6 Conclusion

Tuples provide a clear and concise means to return multiple values from a function. Though C++ has no language constructs for tuples, they can be implemented in standard C++ using templates in a bit inventive fashion. The solution described requires only a few small generic classes and functions (and an up to date compiler) to achieve a completely type safe and efficient tuple implementation.

Up to a certain predefined limit, the proposed tuple template allows tuples to have an arbitrary number of elements, each element being of arbitrary type. The usage is simple and intuitive, semantics being analogous to the pair template in the standard library. The proposed tuple template simplifies the definition and use of functions which return multiple values and is worth of adding to the C++ programmer's basic toolbox.

References

[1] http://www.python.org.

[2] Ashley J. M. and Dybvig R. K.: *An Efficient Implementation of Multiple Return Values in Scheme*, Proceedings of the 1994 ACM Conference on Lisp and Functional Programming, pp. 140-149, Orlando, June 1994.

[3] Paulson L. C.: *ML for the working programmer*, Cambridge University Press, 1991.

[4] *Tuples, routine objects and iterators*, A draft proposal to NICE, http://eiffel.com (link 'Papers').

[5] *International Standard, Programming Languages – C++*, ISO/IEC:14882, 1998.

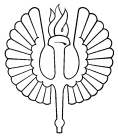
[6] Järvi J.: *Compile Time Recursive Objects in C++*, Proceedings of the TOOLS 27 conference, Beijing Sept. 1998, pp. 66-77, IEEE Computer Society Press.

[7] Myers, N. C.: *A new and useful template technique: 'traits'*, C++ Report, Vol. 7 no 5 pp. 32-35, 1995.

[8] Stroustrup, B.: *The C++ programming language, 3rd ed.*, p. 482, Addison-Wesley, 1997.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi



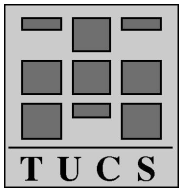
University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**

ML-style Tuple Assignment in Standard C++ — Extending the Multiple Return Value Formalism

Jaakko Järvi



Turku Centre for Computer Science
TUCS Technical Report No 267
April 1999
ISBN 952-12-0434-6
ISSN 1239-1891

Abstract

It has been shown previously that tuples can be implemented with templates in standard C++. This article extends the author's previous work on tuples and multiple return values by defining generic functions and classes to support a kind of a multivalued assignment. The return values of a function returning a tuple can be assigned to separate variables with a single statement. E.g., assuming that foo is a function returning a 3-tuple and that a, b and c are variables of correct type, the expression tie(a, b, c) = foo() assigns the three returned elements to a, b and c.

The presented template definitions result in a typesafe and intuitive tuple formalism, resembling the the tuple constructs in ML, for defining and calling functions having multiple return values. The produced code is efficient. Resource consumption of compiling increases to some extent, but hardly significantly on real programs.

Keywords: generic programming

TUCS Research Group
Algorithmics group

1 Introduction

In a previous article [1], the problem of multiple return values was addressed. It was argued that *tuples*, as found in the programming languages ML[2] or Python[3], provide a concise and intuitive means for returning multiple values from functions. To compensate the lack of direct support for tuples in C++, a set of generic functions and classes implementing a tuple formalism were described in the article. Instead of using reference parameters to pass data out of functions, or writing separate classes for grouping return values, the presented templates definitions make it possible to use tuple types as return values.

We shortly revisit the SVD example from [1]. SVD is a matrix decomposition operation decomposing a matrix into two matrices U, V and one vector S, representing a typical situation, where a function produces several results. The following code snippets show the am. three possibilities for returning the three resulting objects.

```
// 1. Using reference parameters:
void SVD(const Matrix& m, Matrix& U, Vector& S, Matrix& V);
```

```
// 2. Using a separate class:
struct SVD_result {
  Matrix U, V;
  Vector S;
  // constructor, etc.
};
SVD_result SVD(const Matrix& m);
```

```
// 3. Using tuples:
tuple<Matrix, Vector, Matrix> SVD(const Matrix& m);
```

As demonstrated by this example, the prototypes of functions returning multiple values can be defined very naturally using tuples.

Functions returning tuples can called as follows (see [1] for the details of the element access in tuples):

```
Matrix U, V; Vector S;
        ...
tuple<Matrix, Vector, Matrix> result = SVD(aMatrix);
U = get<1>(result); S = get<2>(result); V = get<3>(result);
```

This is rather convenient. However, it would be even more convenient to be able to assign the tuple elements to variables in a single statement. This is

possible, for example, in ML and Python. In numerical programming, a similar construct is used extensively in the scripting language of MATLAB [4]. Using these languages, the SVD example could be written as:

```
(U, S, V) = SVD(aMatrix)       # Python
val (U, S, V) = SVD(aMatrix);  (* ML *)
[U, S, V] = SVD(aMatrix);      % Matlab
```

Though it may not be apparent, this powerful construct is attainable in standard C++ as well. In C++ syntax, the above example can be written as:

```
Matrix U, V; Vector S;
      ...
tie(U,S,V) = SVD(aMatrix);
```

The idea of redirecting return values in the above manner was originally presented in a usenet article by Ian McCulloch. The technique was proposed to be used with functions returning *std::pair* objects, being thus limited to only two elements.

This article shows how the techique can be generalised to tuples of arbitrary length and describes the implementation in detail. The implementation extends the tuple formalism presented in [1]. New class and function tempates are added to allow the simultaneous assignment of elements. The existing definitions require no changes.

2 Assigning tuple elements to variables

How does the above example C++ code work? Fig. 1 depicts the workings of the example graphically. The right-hand side of the assignment expression tie(U,S,V) = SVD(aMatrix) is an ordinary call to a function returning a tuple object. In the left-hand side, tie is a function template. The evaluation of the whole expression comprises of the following steps, where the order of steps 1 and 2 is implementation dependent:

1. The function call SVD(aMatrix) returns a tuple object, the type of which is tuple<Matrix, Vector, Matrix>.

2. tie(U, S, V) is a call to an instantiation of a generic function. This generated function creates and returns a *tier* object, a kind of a tuple as well. The elements of this tier-tuple are references to variables U, S and V.

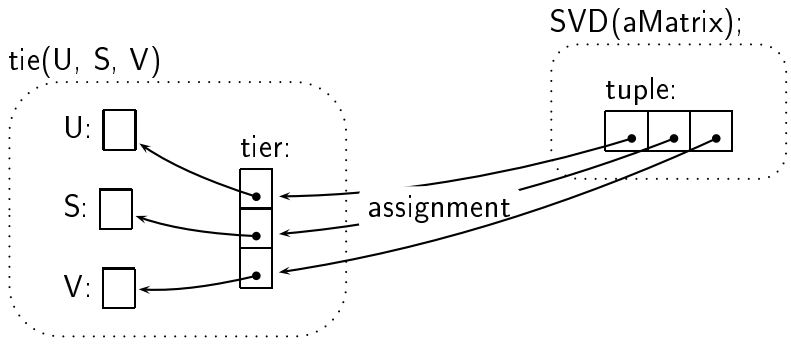


Figure 1: The evaluation of the expression tie(U, S, V) = SVD(aMatrix) illustrated graphically. The leftmost and rightmost dotted frames outline the results of the left-hand and right-hand sides of the expression.

3. The assignment operation assigns the tuple object to the tier object, performing an element-wise assignment from the tuple elements to the tier elements. Since the tier elements are references, the actual destinations of the assignments are the original variables referred to, that is, U, S and V.

4. The temporary tier object is destructed.

2.1 About terminology

The tier acts as an intermediary object *binding* the returned tuple elements to a set of variables, and indeed, as opposed to tying, binding is a more conventional term in the programming language literature. However, the terms *tie* and *tier* were chosen, since the C++ standard library contains a set of binder templates for a different purpose.

3 Tier template definitions

The structure of the tier template definitions correspond to the structure of the tuple template definitions in [1]. The tuple code is repeated in its relevant parts in appendix A.

The backbone of the tuple implementation is the *cons* template, implementing a compile time list structure. Different recursive instantiations of this template provides tuples the ability to store an arbitrary number of elements of arbitrary types. Tiers need the same capability, with the distinction

that the elements are references. For this purpose the ref_cons template, a modifcation of the cons template, is defined:

```
struct nil {};

template <class HT, class TT> struct ref_cons {
  HT& head; TT tail;
};

template <class HT> struct ref_cons<HT, nil> { HT& head; };
```

The empty nil class represents the end-mark of the list. Now, for example the instantiation

```
ref_cons<Matrix, ref_cons<Vector, ref_cons<Matrix, nil> > >
```

defines a type containing a Vector reference and two Matrix references as member variables. To evade this unwieldy syntax, the ref_cons template is not instantiated directly. The tier template defines a more usable interface and inherits a suitable ref_cons instantiation. The tier template allows a variable number of template arguments, up to a predefined limit, using the technique presented in [1]. For terseness, this limit is chosen to be four here, but it is straightforward to increase it by extending the template parameter lists:

```
template <class T1, class T2 = nil, class T3 = nil, class T4 = nil>
struct tier : public tier_to_ref_cons<T1, T2, T3, T4>::U {...};
```

The tier_to_ref_cons template is a recursive traits class [5] mapping the tier template parameters to the correct ref_cons instantiation.

```
template <class T1, class T2, class T3, class T4>
struct tier_to_ref_cons {
  typedef ref_cons<T1, typename tier_to_ref_cons<T2, T3, T4, nil>::U > U;
};

template <class T1> struct tier_to_ref_cons<T1, nil, nil, nil> {
  typedef ref_cons<T1, nil> U;
};
```

These definitions allow, for example, the previous example to be written as tier<Matrix, Vector, Matrix>. For a more verbose description of recursive type mappings, see [1, 6].

3.1 Constructing tiers

The tier and ref_cons constructors are structurally equivalent to the tuple and cons constructors. First the tier constructor:

```
template <class T1, class T2 = nil, class T3 = nil, class T4 = nil>
struct tier : public tier_to_ref_cons<T1, T2, T3, T4>::U {
        ...
  tier( T1& t1, T2& t2=wrap_ref<T2>(),
        T3& t3=wrap_ref<T3>(), T4& t4=wrap_ref<T4>())
    : tier_to_ref_cons<T1, T2, T3, T4>::U(t1, t2, t3, t4) {}
        ...
```

The constructor takes four parameters (four being the upper limit chosen), and delegates them to the base class constructor defined in section 3.1.2. The intention is not to require four arguments in each constructor call. The number of arguments accepted by the constructor should equal the number of actual (non-nil) elements in the tier, ranging from one to the maximum number of allowed elements in a tuple. This is achieved with the wrap_ref template used as default arguments.

3.1.1 Default arguments

The actual arguments to the tier constructor are the variables to which the tuple elements should be assigned. There are thus no reasonable default values for the actual arguments. On the other hand, default values are needed for the unspecified arguments of type nil. Since the number of nil-arguments is not know beforehand, the default argument expressions should return a suitable default value if the type of the argument is nil, and generate a compile-time error for other types. The following definitions fulfill these requirements:

```
template <class T> struct ct_error {};

namespace { nil dummy_nil; }

template <class T> struct wrap_ref_struct {
  static T& f() {
    return ct_error<T>::missing_argument; // Yields a compile time error
  }
};

template <>
struct wrap_ref_struct<nil> { static nil& f() { return dummy_nil; } };

template <class T> T& wrap_ref() { return wrap_ref_struct<T>::f(); }
```

The separation between nil and non-nil template arguments can not be accomplished directly with the wrap_ref template, since the template parameter must be explicitely specified, prohibiting thus specialisations. Consequently, the function template wrap_ref is just an interface, delegating the task to a static function f of a template class wrap_ref_struct, which can be specialized.

In the specalisation wrap_ref_struct<nil>, the static function f returns correctly a reference to a nil object. Since the parameters to the tier constructor are passed as references, even the nil parameters must refer to some existing object, hence the dummy_nil object. It is placed in an unnamed namespace to make it local to a translation unit. As becomes clear in the sequel, the need for the dummy_nil object is temporary, the final tier object does not contain any references to this object.

The function f in the primary template is instantiated whenever a default argument is needed for any other type than nil. This introduces a compile time error, as required, by referring to a non-existing member of a class template.

3.1.2 Constructing ref_cons objects

The tier constructor passes its parameters, nil and non-nil ones, directly to the constructor of its base class:

```
template <class HT, class TT> struct ref_cons {
        ...
  template <class T2, class T3, class T4>
    ref_cons( HT& t1, T2& t2, T3& t3, T4& t4)
      : head(t1), tail(t2, t3, t4, dummy_nil) {}
        ...
};

template <class HT> struct ref_cons<HT, nil> {
        ...
  ref_cons(HT& t1, nil, nil, nil) : head (t1) {}
        ...
};
```

This constructor recursively initializes each element of the tier. At each recursive call, a reference to the first argument is stored in the current head and the remaining arguments are passed to the constructor of the tail.

3.2 Tie function templates

The above definitions allow the definition of tier types and construction of tier objects. For example, tier<Matrix, Vector, Matrix>(U, S, V) creates a

tier holding references to U, S and V. The objective was, however, to be able to construct tier objects using a simpler syntax tie(U, S, V), avoiding thus the explicit specification of the element types. This can be attained by wrapping the constructor call inside a function template, analogously to the *make_pair* template in the standard library, and letting the compiler deduce the argument types. However, this is somewhat problematic.

The C++ standard bans default template arguments in function templates. Neither do default arguments have any effect on the template argument deduction [7, section 14.8.2.4]. Due to these restrictions, there seems to be no way to define a tier function template, which would be generic with respect to the number of arguments. Hence, a separate function template must be written for each allowed tier length. Albeit not very elegant, this is however perfectly feasible. In any case, the maximum number of elements in tuples is not very high. As functions with very long parameter lists tend to be error-prone and impractical, same is true for very long tuples as return values. The need for tuples longer than few dozens of elements is hardly justified. Consequently, the amount of code duplication is quite tolerable. Futhermore, as the code duplication does not affect the client code in any way, except of course by providing the desired functionality, the duplication remains merely as a source of discontent to the author.

As an example, the two and three argument tie function templates are defined as follows:

```
template<class T1, class T2>
inline tier<T1, T2> tie(T1& t1, T2& t2) {
  return tier<T1, T2>(t1, t2);
};

template<class T1, class T2, class T3>
inline tier<T1, T2, T3> tie(T1& t1, T2& t2, T3& t3) {
  return tier<T1, T2, T3>(t1, t2, t3);
};
```

4 Assigning tuples to tiers

We now have means to conveniently create tier objects, but the assignment operators from tuples to tiers are yet to be defined. As stated in section 2, these operators perform an element-wise assignment from tuple to tier elements.

Separate assignment operations are required in the tier template, as well as in the ref_cons templates. Each of these assignment operations are defined

as member templates. The parameter in the tier assignment is constrained to be a tuple.

```
template <class T1, class T2 = nil, class T3 = nil, class T4 = nil>
struct tier : public tier_to_ref_cons<T1, T2, T3, T4>::U {
        ...
  template <class V1, class V2, class V3, class V4>
  void operator=(const tuple<V1, V2, V3, V4>& t) {
    tier_to_ref_cons<T1, T2, T3, T4>::U::operator=(t);
  }
        ...
```

The tuple is redirected as such to the assignment operator of the base class, performing the actual assigning. Note, that even though the definition does not restrict the types of the tuple elements in any way, the assignment is typesafe. An attempt to assign a tuple with incompatible element types is caught at some of the assignments of the nested members of the ref_cons base class.

```
template <class HT, class TT> struct ref_cons {
        ...
  template <class HT2, class TT2>
  void operator=(const cons<HT2, TT2>& u) { head=u.head; tail=u.tail; }
        ...
};

template <class HT> struct ref_cons<HT, nil> {
        ...
  template <class HT2>
  void operator=(const cons<HT2, nil>& u) { head = u.head; }
        ...
};
```

The assignment operations in the ref_cons templates assign the elements recursively. Again, the parameters are only restricted to be of type cons, the element types are not confined explicitly. However, the assignment of the heads fails, if the types HT and HT2 are incompatible. In sum, the assignment leads to a compile time error, if the tuple and tier are not of the same length and if the types of the corresponding elements are not assignable.

4.1 Performance considerations

4.1.1 Runtime cost

As shown in [1], there is no performance penalty in returning tuples compared with returning explicitely defined classes from functions. The introduction of

tiers does not change this, since the use of tiers induce no changes to functions returning tuples. However, the code for calling functions having multiple return values is different, as tiers add a new mechanism for redirecting the resulting values. In this section, we focus on the cost of this redirection. Consider the code:

```
// A, B, C, D are some classes
tuple<A, B, C, D> foo();
        ...
A a; B b; C c; D d;
        ...
tie(a, b, c, d) = foo();
```

The tie function creates the tier object. Since the tie function, tier constructor, and the constructors of the inherited ref_cons classes are all inlined, the effect of the tier construction process is roughly equivalent to:

```
A& a_ = a; B& b_ = b; C& c_ = c; D& d_ = d;
```

foo is a function returning a tuple. Right after foo has been executed, the resulting tuple resides somewhere in the memory (generally near the top of the stack) available for furher use. Now the assignment operator of the tier object is called. The operator assigns the elements of the tuple to the tied variables. Again, the assignment operations are all inlined, eliminating all overhead and leaving only the calls to the assignment operations of the individual elements. This step corresponds to the statements:

```
a_ = get<1>(foo_r); b_ = get<2>(foo_r);
c_ = get<3>(foo_r); d_ = get<4>(foo_r);
```

foo_r stands for the temporary tuple object returned from foo. As the left hand sides are references, the statements are assignments to a, b, c and d.

No destructor was defined for the tier template, the compiler generated destructor is trivial [7, chapter 12.4] and thus yields no code.

In sum, the cost arising from the tying mechanism is the cost of creating one reference variable and performing one assignment for each element of the tuple. The details are obviously dependent on the compiler, but this is more or less the behaviour one would expect from an optimising C++ compiler.

These assumptions were tested by studying the compiled code of several test programs. A diverse set of tuples and tiers was used: the elements ranged from built-in types to rather complex classes and the length of tuples was varied. Two compilers were used: KAI C++ (verion 3.3c) and Egcs (release 1.1.1).

Both compilers confirmed the expected behaviour. The code produced by Egcs was constistently in accordance with the above description. KAI C++ could in some cases even avoid allocating stack space for the reference variables by locating them into registers; the extra assignment was still performed. As mentioned in [1], KAI C++ was not capable of inlining very long tuples entirely, unless exceptions were turned off by a compiler switch.

Due to the extra assignment, there is some overhead in the tier mechanism compared with using reference parameters to pass data out from a function. For lightweight objects with low assignment cost the overhead is most likely not significant. For large objects with high copy or assignment semantics, the extra cost may be notable. However, returning such objects, inside tuples or otherwise, should be avoided anyhow. Instead, large objects ought to be wrapped inside auto pointers, reference counted pointers etc. providing cheap copy and assignment. Hence, tiers can be applied without a significant performance penalty in vast majority of the cases where multiple values need to be returned.

4.1.2 Compilation cost

Compiling code containing tuple and tier definitions is somewhat slower and requires more memory than compiling corresponding code not containing these definitions. There is discussion on the effect of tuples to compilation time and memory consumption in [1], showing that this effect is clear but on real programs probably not significant. The introduction of tiers to the multiple return value formalism does not change this, which was confirmed with two compilation tests.

The compilation times of tier constructs were compared with the compilation times of corresponding definitions using reference parameters to 'return' multiple values. This is a natural choise, since in both cases there is a set of existing variables to which the results should be bound. To measure the direct compilation speed difference of the constructs, five pairs of test programs were generated. Each pair contained a program using the tier mechanism and a functionally equivalent program using reference parameters in multivalued returns. The tier programs comprised solely of functions returning tuples and calls to these functions using the tier mechanism, hence the compilation times reflect the direct compilation cost difference between the two alternatives for returning multiple values.

The results are shown in Fig. 2, indicating clearly the slower compilation times of tier constructs. However, the compilation of parameter passing constructs is only a small part of the whole compilation process. Hence, to estimate the effect on the compilation times of real C++ programs, an existing

Tuple length	Egcs (T_{tier}/T_{ref})	KAI C++ (T_{tier}/T_{ref})
2	12.9	23.09
4	13	26.27
8	14.83	32.81
16	16.42	55.81
32	22.72	71.43

Figure 2: The relative compilation time of programs using tuples and tiers to equivalent programs using reference parameters to pass multiple values from a function (T_{tier} = compilation time of tuple implementation, T_{ref} = compilation time of reference parameter implementation). To eliminate the effect of any constant costs of the compilation process, the compilation time of a program containing an empty main function was subtracted from the results. The times are not comparable across compilers.

program was selected to represent an 'average' C++ program. The program included a few standard headers and used standard generic containers and algorithms quite frequently. A reasonable amount of tier constructs were added to the program. As a result, the program comprised of 150 functions, 20% of which returned tuples and 20% used the tier mechanism to call the tuple returning functions. The length of tuples ranged from 2 to 8, shorter tuples being more common than longer ones. The lenghts and percentages were selected based on the authors estimations about the frequency of functions with multiple return values. The program is intended to represent quite an extensive use of the tier mechanism.

The compilation of this program was compared with a functionally equivalent program using reference parameters instead of the tier mechanism. The increase in compilation time was 5% in Egcs and 11% in KAI C++. Memory consumption increased more, 27% (Egcs) and 22% (KAI C++).

5 Conclusion

An implementation of a generic tuple class was introduced in [1] to alleviate the definition and calling of functions with multiple return values in C++. This article extends the tuple implementation, describing the tier mechanism enabling a kind of a multivalued assignment, i.e., a means to assign the elements of a tuple to separate variables with a single statement.

Compared with using reference parameters as output parameters, the presented mechanism has a small performance penalty, basically an extra assignment is needed for each tuple element. Compiling code using the tier

mechanism requires more resources than conventional code. The tests performed suggest that these increases are probably not significant, though possibly notable with very extensive usage of the mechanism.

Tuples and tiers together provide a convenient and intuitive formalism for defining and calling functions with multiple return values. The techniques are efficient and do not compromise type safety.

A Relevant parts of tuple implementation

```
struct nil {};

template<class HT, class TT> struct cons {
  HT head; TT tail;
  template <class T2, class T3, class T4>
  cons(const HT& t1, const T2& t2, const T3& t3, const T4& t4)
    : head(t1), tail(t2, t3, t4, nil()) {}
};
template <class HT> struct cons<HT, nil> {
  HT head;
  cons(const HT& t1, const nil&, const nil&, const nil&) : head (t1) {}
};

template <class T1, class T2, class T3, class T4>
struct tuple_to_cons {
  typedef cons<T1, typename tuple_to_cons<T2, T3, T4, nil>::U > U;
};
template <class T1>
struct tuple_to_cons<T1, nil, nil, nil> { typedef cons<T1, nil> U; };

template <class T> inline T wrap() { return T(); };

template <class T1,class T2=nil,class T3=nil,class T4=nil>
struct tuple : public tuple_to_cons<T1, T2, T3, T4>::U {
  tuple( const T1& t1=wrap<T1>(), const T2& t2=wrap<T2>(),
         const T3& t3=wrap<T3>(), const T4& t4=wrap<T4>())
    : tuple_to_cons<T1, T2, T3, T4>::U(t1, t2, t3, t4) {}
};
```

References

[1] Järvi J.: *Tuples and multiple return values in C++*, TUCS Techincal Report No 249, http://www.tucs.fi/publications.

[2] Paulson L. C.: *ML for the working programmer*, Cambridge University Press, 1991.

[3] http://www.python.org.

[4] The Mathworks Inc. (http://www.mathworks.com): *Using MATLAB, Version 5*, 1997.

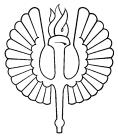
[5] Myers, N. C.: *A new and useful template technique: 'traits'*, C++ Report, Vol. 7 no 5 pp. 32-35, 1995.

[6] Järvi J.: *Compile Time Recursive Objects in C++*, Proceedings of the TOOLS 27 conference, Beijing Sept. 1998, pp. 66-77, IEEE Computer Society Press.

[7] *International Standard, Programming Languages – C++*, ISO/IEC:14882, 1998.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi



University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**

C++ Function Object Binders Made Easy

Jaakko Järvi *

Turku Centre for Computer Science
Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland
jaakko.jarvi@cs.utu.fi

Abstract. A novel argument binding mechanism that can be used with STL algorithm invocations is proposed. Without using any adaptors, binding can be applied directly to pointers to nonmember functions, pointers to const and nonconst member functions and STL function objects. The types and number of arguments in the functions to be bound can be practically arbitrary; argument list lengths up to few dozens of elements can be supported.
The unbound arguments are expressed as special placeholders in the argument list; they can appear for any argument position. Hence, binding sites preserve the resemblance to the function prototype of the underlying function, leading to simple and intuitive syntax.
Binding can be applied recursively. This results in a versatile function composition mechanism. The binding mechanism is efficient in the sense that it induces very little or no runtime cost.

1 Introduction

The Standard Template Library (STL) [1], now part of the C++ standard library [2], is a generic container and algorithm library. STL algorithms are function templates operating on container elements via iterators and *function objects*. Any C++ construct which can be called with the ordinary function call syntax is a function object. This includes pointers and references to nonmember functions and static member functions, and class objects with a function call operator. Nonstatic member functions are not function objects, but the STL includes a set of *function adaptors* for creating function objects from member functions.

Adaptable function objects are a subset of function objects. They are class objects which, in addition to the function call operator, define a certain set of types. New adaptable function objects can be created from adaptable function objects using function adaptors. For example, function objects can be projected by binding one argument to a constant. Some STL implementations [3] add function composition objects and other adaptor extensions to the standard. Superficially, this seems to provide a great amount of flexibility for adapting functions

* This work has been supported by the Academy of Finland, grant 37178. The author is grateful to Harri Hakonen and Daveed Vandevoorde for their valuable comments on the manuscript of this paper.

to satisfy commonly encountered needs. However, a closer look reveals that this is not the case.

The function objects accepted by STL algorithms are either *nullary*[1], *unary* or *binary*. Standard adaptors accept only unary or binary function objects. Consequently, all functions with more than two parameters, and all member functions having more than one parameter, cannot be passed as function objects to STL algorithms. To be able to pass such functions to STL algorithms, explicit function object classes must be written. Often this results in considerable programming overhead—especially when only a simple operation or algorithm needs to be supported. This inconvenience in turn discourages the use of an otherwise compact and intuitive functional programming style, and not infrequently the STL invocation is expanded by hand.

For example, consider the following two functions for computing values of exponential and Gaussian distributions:

```
double exponential(double x, double lambda);
double gaussian(double x, double mean, double standard_deviation);
```

The computation of the values of the exponential distribution for a set of points (in vector x) can be programmed as follows:

```
vector<double> x, result;    double lambda;
        ...
transform(x.begin(), x.end(), result.begin(),
            bind2nd(ptr_fun(exponential), lambda));
```

The ptr_fun wrapper creates an adaptable binary function object from the exponential function and bind2nd binds the second argument to lambda. The transform algorithm calls this unary function object for each element in x and places the result in result.

We should expect the code for computing Gaussian distribution values to be analogous. However, it turns out that an explicit function object class is needed:

```
vector<double> x, result;    double mean, std;
        ...
class gaussian_caller {
  double mean_, std_;
public:
  gaussian_caller(double mean, double std) : mean(mean_), std(std_) {};
  double operator()(double x) const { return gaussian(x, mean, std); }
}
transform(x.begin(), x.end(), result.begin(), gaussian_caller(mean, std));
```

This is a consequence of the ptr_fun adaptor only being defined for functions taking fewer than three arguments. To overcome this limitation, it is possible to define more ptr_fun templates to cover functions with longer argument lists.

[1] A function with no arguments.

Unfortunately, this is not enough, since the binder adaptors are defined for binary function objects only. Hence, n binders (bind1st–bindNth) would be needed for each argument list of length n. Even with all these $k^2/2$ templates, k being the longest allowed argument list length, the outcome is not particularly intuitive. It becomes difficult to see how the gaussian function is actually called:

```
transform(x.begin(), x.end(), result.begin(),
          bind2nd(bind3rd(ptr_fun(gaussian),std),mean));
```

This article proposes a more general binding mechanism. Instead of specifying the index of the argument to bind, special *placeholder* objects are used in the argument list. For example, the previous example becomes:

```
transform(x.begin(), x.end(), result.begin(), bind(gaussian, free1, mean, std));
```

The role of the placeholder free1 is to specify the varying argument to be left unbound. Other arguments are bound to the values given. The binder invocation syntax preserves a direct resemblance to the original function prototype, revealing instantly which parameters are bound. This mechanism bears similarities to the built-in binding mechanism of the programming language Theta [4], as well as with *agents*, a recently proposed extension to Eiffel [5]. The technique is also related to *currying*, the partial function application mechanism in the functional programming domain.

This article describes the functionality and design of the generic *Binder Library*, BL in the sequel. The BL allows argument binding in the above style for (almost) any C++ function. The library can be downloaded from the address *www.cs.utu.fi/BL*.

2 Binder library functionality

Binding is an operation which creates a k-argument function object, a *binder object*, from an n-argument *bindable function object*, such that $k \leq n$. Some of the arguments of the original function object are bound to fixed values, the remaining unbound arguments are called *free*.

2.1 Bindable function objects

Binding can be applied to nonmember functions and to static member functions, as well as to pointers to such entities. Pointers to const and nonconst member functions are bindable as well. Other function objects, i.e., class objects with a function call member operator, are bindable if they contain a specific set of member typedefs. For smooth integration with the STL, function objects adhering to the adaptable function object requirements of the STL are bindable. Constructors are not bindable.

The STL has special wrapper templates for creating adaptable function objects from nonmember function (ptr_fun) and member function (mem_fun,

mem_fun_ref) pointers. They provide a uniform call syntax for different types of functions: a wrapped n argument member function can be called with $n + 1$ arguments, where the first argument is a reference or pointer to the object whose member is called. In the BL such wrappers are not needed. (A similar mechanism is used but it is hidden from the client.) Hence, binding can be applied directly to pointers to member and nonmember functions.

2.2 Function argument list length

As explained in section 1, STL style binders cannot easily be applied to functions with more than two arguments. The binding mechanism using placeholders, in turn, does not have this restriction. However, the implementation requires a set of template definitions for each supported argument list length. This imposes a predefined upper limit for the number of allowed arguments. The limit is not particularly restrictive, since argument lists up to a few dozen elements can be supported—the size of the library grows linearly with the supported argument list length.

2.3 Argument binding

A binder object is created with a call to an overloaded generic bind function. The first argument in this call is the bindable function object, henceforth called the *target function*. The remaining arguments correspond to the argument list of the target function.

The argument list can contain free arguments, which are specified with placeholder objects. Since there are no STL algorithms accepting function objects with more than two arguments, the BL defines just two placeholder objects. These are called free1 and free2 and refer to the first and second argument of the function call operator in the resulting function object. Depending on which placeholders are used, this function call operator is either nullary, unary or binary.

Consider the following examples, where Op is a four-argument bindable function object callable with arguments of some types A, B, C and D. Further assume that a, b, c and d are variables of these types, respectively.

```
bind(Op, a, b, c, d);
bind(Op, a, free1, c, d);
bind(Op, free1, b, c, free2);
bind(Op, free2, free2, free1, free1);
```

The first line creates a nullary function object fixing all arguments. The second invocation of bind creates a unary function object taking one argument of type B. The third line results in a binary function object with the first argument of type A and second of type D. The last use of bind illustrates how several free arguments can be unified. In this case A and B as well as C and D must be compatible types. The call results in a function object taking two variables. The first is of type C, second of type A. An invocation with, say, c and a, results in a

call Op(a, a, c, c) of the original function object. Therefore, a must be implicitly convertible to B and c to D.

To summarise, the argument list can contain an arbitrary number of both types of placeholders, but if it contains one or more free2 placeholders, it must contain at least one free1 placeholder. A violation of this rule leads to a compile time error. This is reasonable, since the omission of free1, while free2 is present, would result in a function object having the second argument but not the first.

2.4 Argument types

The parameter types of target functions can be arbitrary[2]. However, references to nonconst objects and to objects which cannot be copied require special treatment.

Arguments to bind functions are passed as reference to const. Thus, if a parameter of type reference to nonconst object is bound, the actual argument must be wrapped inside a special object, which creates a temporary const disguise for the argument. This is achieved with a simple function call. For example:

```
void up_and_down(int& i, int& j) { i++; j--;}
list<int> a_list;    int counter = 0;
        ...
for_each(a_list.begin(), a_list.end(), bind(up_and_down, ref(counter), free1));
```

The example decrements each element of a_list by one and increments counter by one n times, n being the number of elements in a_list. Since counter is bound to the nonconst reference parameter i, it is wrapped using the generic ref function. Although the parameter corresponding to the free argument is a reference to a nonconst type as well, it need not be wrapped. The wrapping mechanism is safe with respect to constness: a const reference cannot be wrapped.

The original motivation for requiring the use of the ref wrapper was the desire to avoid combinatorial explosion of template definitions. However, it is semantically beneficial as well. As the example above demonstrates, side effects in bound arguments quickly lead to code that is hard to comprehend. Wrapping forces the programmer to explicitly state that an argument is susceptible to side effects.

By default, the binder object stores copies of the bound arguments. If an argument is of a type which cannot be copied, another wrapper is needed: cref instructs the binder object to store a reference to the argument instead of a copy. Array types are exceptions. Although they can not be copied, wrapping is not necessary: a reference to const array type is stored by default. This ensures that string literals can be used directly as bound arguments. The cref wrapper is used similarly to the ref wrapper.

[2] Volatile qualified types are not supported in the current version of the BL.

2.5 Member functions

When binding member functions, the object for which the member function is to be called, is the first argument after the target function. This *object argument* can be a reference or pointer to the object; the BL supports both cases with a uniform interface. Similarly, if the object argument is free, the sequence can contain either pointers or references. For instance:

```
bool A::f(int); A a;
vector<int> ints;    vector<A> refs;    vector<A*> pointers;
        ...
find_if(ints.begin(), ints.end(), bind(&A::f, a, free1));
find_if(ints.begin(), ints.end(), bind(&A::f, &a, free1));
        ...
find_if(refs.befin(), refs.end(), bind(&A::f, free1, 1));
find_if(pointers.begin(), pointers.end(), bind(&A::f, free1, 1));
```

The first two calls to find_if are equivalent. In the first one, A::f is called using the .* -operator, whereas in the second the same member is called through the –>* -operator. The latter two find_if invocations both find the first A for which A::f returns true. The .* -operator is used in both cases, the library automatically dereferences object arguments of pointer types.

The call mechanism is safe with respect to constness. A nonconst member function can only be called via a reference or pointer to a nonconst object. This holds whether the object argument is bound or free. Note that the ref wrapper is not needed for the object argument. This is a deliberate design choice. It is understood that the object's state may be changed when a nonconst member function is called—there is no need to explicitly state it.

2.6 Function composition with binders

Binding can be applied recursively, thereby enabling function composition. Consider the following example:

```
void canvas::point(double x, double y, colour c);
canvas* canv;    vector<double> x, y;
for_each(x.begin(), x.end(), y.begin(),
        bind(&canvas::point, canv, free1, free2, black));
```

point is a function for drawing points on a canvas using some colour. The for_each invocation draws the points given by the coordinates in vectors x and y. Note that this two-sequence for_each algorithm is not part of the C++ standard library, it is however straightforward to write [6, p. 532].

Now, to represent the y coordinates in logarithmic scale, the standard library function double log(double) can be bound as follows:

```
for_each(x.begin(), x.end(), y.begin(),
        bind(&canvas::point, canv, free1, bind(log, free2), black));
```

The resulting binder object invokes canv->point(*iter1, log(*iter2), black) at each iteration, where iter1 and iter2 are the iterators provided by for_each: they point to the elements of x and y. Hence, a nested binder object defers the target function call and acts as the inner function in a function decomposition.

The number of nested recursive bindings can be arbitrary. With respect to placeholders, the argument lists of all nested binders are treated as one concatenated list. This means that the requirements stated at the end of section 2.3 must hold for the concatenated argument list, not for any of the individual lists.

2.7 Nullary functions

For completeness, nullary functions can be bound as well. This may seem superfluous, but with function composition, it is a powerful feature. For instance, the following code creates a vector of 100 random number pairs:

```
vector<pair<int, int> > pvec;
generate_n(back_inserter(pvec), 100,
           bind(&make_pair<int, int>, bind(rand), bind(rand)));
```

3 Library design

This section describes the general library design and discusses the key programming techniques used in its implementation. Certain technical details have been omitted from the presented code for clarity.

To begin with, some parts of the binder library cannot be written generically with respect to the length of the target function parameter list. A set of templates are repeated with slight modifications for each supported argument list length. A great deal of this repetition can be avoided using *tuples*.

3.1 Tuples

The binder library uses a set of templates comprising a tuple abstraction [7]. These template definitions are rather intricate and not presented here. Nevertheless, their usage is intuitive.

The tuple template is basically a generalisation of the pair template in the standard library. It can be instantiated to contain an arbitrary number (up to some predefined limit) of elements of arbitrary types. E.g., tuple<int, string, A> is a valid tuple type, corresponding to a class having three member variables of types int, string and A. As is the case of standard pairs, tuples can be constructed directly or using make_tuple (cf. make_pair) helper templates:

```
tuple<int, string, A>(1, string("foo"), A());
make_tuple(1, string("foo"), A());
```

The elements can be accessed in a generic fashion with the syntax get<N>(x), where x is some tuple and N is an integral constant stating the index of the element. The get function template is a *template metaprogram* [8], which resolves a reference to the given element of a tuple at compile time.

The types of the tuple elements can be expressed generically as well: the expression tuple_element<N, Y>::type gives the type of the Nth element of a tuple type Y. This expression is a kind of a *type function* from the constant N and type Y to the element type.

The implementation is based on compile time lists [9, 10], which are recursively instantiated templates. Consider, for example, how the pair instantiation pair<int, pair<string, pair<A, nil> > > could represent the preceding tuple. The compile time lists in the BL are structurally similar, but they are not based on the pair template.

3.2 The bind function templates

The bind function templates define the binding interface. They are repeated for each argument list length. For example, the four-argument bind function is defined as:

```
template <class Target, class A1, class A2, class A3>
inline binder<Target, 3, tuple<A1, A2, A3> >
bind(Target fun, const A1& a1, const A2& a2, const A3& a3) {
  return binder<Target, 3, tuple<A1, A2, A3> >
    (fun, make_tuple(a1, a2, a3));
};
```

These functions group the actual arguments to the target function into a tuple and create a binder object. There are two overloaded bind function templates for each argument list length because nonconst target member functions must be handled differently.

3.3 Binder objects

An important goal in the design of the BL was to minimise the code repetition. This has considerably impacted the structure of the binder classes. Binder objects are instances of class types that fit in the four-level class hierarchy illustrated in Fig. 1. Each level in the hierarchy encapsulates some task orthogonal to the tasks in the other levels. This reduces the amount of partial specialisations, since only one property must be taken into consideration at a time. It also avoids the need to repeat the member and type definitions which are generic with respect to the argument list length.

Target function The base class target_function is a generalisation of the STL templates unary_function and binary_function:

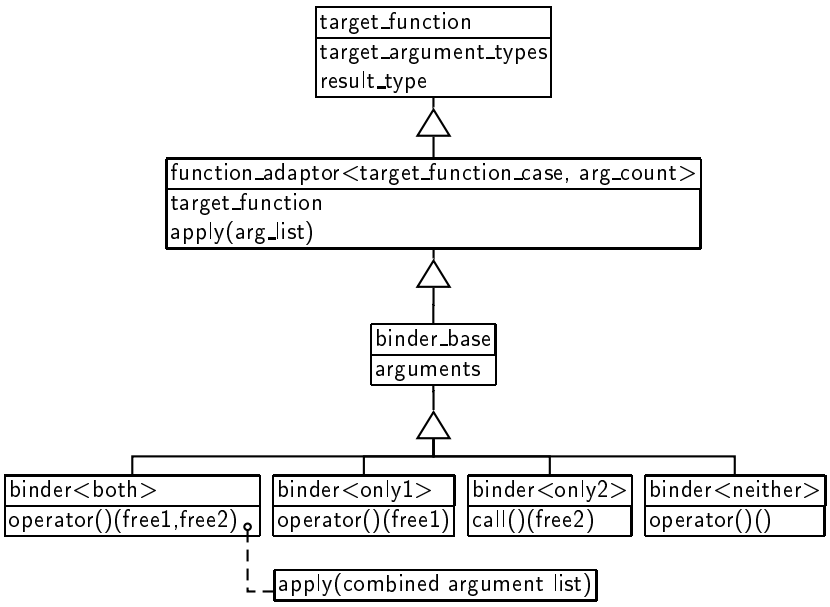


Fig. 1. Binder class hierarchy. The binder template arguments represent the intuitive interpretation of the specialisation criteria. The combined argument list in the call to the inherited apply function refers to the merged argument list where the placeholders have been substituted with the parameters of the function call operator.

```
template<class Args, class Result> struct target_function {
  typedef Args target_argument_types;
  typedef Result result_type;
};
```

It contains type definitions which specify the argument types and the result type of the target function. These type definitions are necessary in various type functions presented below. The same template covers all target function argument lengths, since the template is instantiated with Args substituted by a tuple type.

Function adaptors The purpose of the function adaptors is to unify the function invocation syntax for different types of functions. For each argument length, there are four specialisations of the function_adaptor template. They cover const and nonconst member functions, nonmember functions and other function objects.

Each specialisation contains an appropriately typed data member for storing the target function object and defines the apply function, which provides the uniform call syntax. Another task of the function adaptor templates is to de-

compose the type of the target function and forward the result and argument types to the target_function template.

For example, the primary template and the function adaptor for pointers to three-argument nonmember functions are defined as follows:

```
template <class Target, int ArgsCount> class function_adaptor;
        ...
template <class A1, class A2, class A3, class Result>
class function_adaptor<Result (*)(A1, A2, A3), 3>
  : public target_function<tuple<A1, A2, A3>, Result> {
  Result (*ptr)(A1, A2, A3);
public:
  explicit function_adaptor(Result (*x)(A1, A2, A3)) : ptr(x) {}
  Result apply(A1 a1, A2 a2, A3 a3) const { return (*ptr)(a1, a2, a3); }
};
```

Binder base The binder_base template has one member variable: the tuple containing the actual arguments of the bind call. This is the sole purpose of binder_base. One generic definition suffices for all argument lengths.

Binders The classes described in the preceding presentation are for internal use in the library, whereas the binder template provides the function call operator to be called from client code. For each target function argument list length, there are four bind specialisations. Which specialisation is instantiated depends on the composition of placeholders in the actuals of the bind invocation. For example, if free1 and free2 are both present, a specialisation defining a binary function call operator is instantiated.

The variables free1 and free2 are defined by the library and are of types placeholder<1> and placeholder<2> respectively. The objects themselves are not important but their types are. These types serve as tags that can be localised from the parameter lists using type functions.

The primary binder template is defined as follows:

```
template<class Target, unsigned int N, class Args,
         bool Free1 = find_free<1, Args>::value,
         bool Free2 = find_free<2, Args>::value>
class binder;
```

The first parameter is the target function type, the second the number of arguments in the target function and the third a tuple type representing the actual argument types deduced in the bind function template. The values of the two bool template parameters Free1 and Free2 indicate whether placeholder<1> or placeholder<2> types are present in Args. These values are deduced with the find_free type functions specified as default arguments.[3]

[3] In general, such use of default template arguments is a convenient technique. It can be used to specialise templates with respect to some property (of template arguments) that is not directly usable as a specialisation criterion.

One particular binder specialisation, for the case of a three-argument parameter list including both types of placeholders, is defined as follows:

```
template<class Target, class Args>
class binder<Target, 3, Args, true, true>
  : public binder_base<Target, 3, Args> {
public:
  typedef binder_base<Target, 3, Args> inherited;
  typedef typename inherited::target_argument_types TA;
  typedef typename deduce_free<1, TA, Args>::type first_argument_type;
  typedef typename deduce_free<2, TA, Args>::type second_argument_type;
  explicit binder(const Target& fun, const Args& a) : inherited(fun,a) {}
  typename inherited::result_type operator()
    (first_argument_type a, second_argument_type b) const {
      return apply( choose(get<1>(args), a, b),
                    choose(get<2>(args), a, b),
                    choose(get<3>(args), a, b));
  }
};
```

This covers the case where Free1 and Free2 template parameters both evaluate to true and thus the specialisation defines a binary function call operator. The result type of this function is the result type of the target function, which is defined in the inherited instantiation of target_function.

The argument types are more intricate because they must be deduced by comparing the types in the actual argument type tuple Args and the parameter types of the target function (inherited::target_argument_types). The type function deduce_free<N, Tuple1, Tuple2>::type defines these deductions. Slightly simplified, the first argument type is resolved by locating a placeholder<1> type from Tuple1 and then selecting the corresponding element in Tuple2. The second argument type is deduced in a similar way. These type definitions provide a function call operation with the correct prototype—the correct functionality is achieved with the choose function templates. Their task is to provide the right arguments to the inherited apply function, which invokes the target function. The args tuple contains the actual arguments—some of which are placeholders—of the bind call. Within each argument position, the choose templates select which argument is redirected to apply. For placeholders, a or b is chosen. Otherwise the bound argument, an element of the args tuple, is used.

The choose functions merely return one of their arguments. The types of the arguments determine which argument is returned. The code below shows the definitions of two particular choose function templates.

```
template<class T1, class T2, class T3>
inline T1& choose(T1& a, T2&, T3&) { return a; }

template<class T2, class T3>
inline T2& choose(const placeholder<1>& a, T2& b, T3&) { return b; }
```

3.4 Library design summary

In the interest of brevity, several important issues were omitted from the preceding description. The implementation of reference wrapping and of recursive binding was not described. The type functions were not shown in detail. How to achieve compatibility with STL function objects or ensure correct usage of free arguments were not addressed either. Crucial techniques for avoiding reference to reference situations in binder instantiations were not explained. Nevertheless, the library contains solutions to these technical aspects and provides the functionality described in section 2.

The library consists entirely of template definitions. It has a fixed as well as variable size part. The fixed size part is approximately 700 lines of code. The variable size part grows linearly with the maximum argument list length supported for target functions. Each supported argument list length requires approximately 100 lines of code and that code can be generated mechanically.

4 Performance considerations

An important goal in the design of the STL was to provide a high level of abstraction without sacrificing efficiency [1]. The BL is compatible with this objective. The functions in the BL only redirect arguments and function calls. Every function in the library is inlined so that commercial-grade compilers can eliminate any overhead arising from these redirections.

In various tests performed with the gcc C++ compiler (version 2.95)[11], the performance of STL algorithms using BL style binder objects was nearly identical to algorithms using standard STL binders. Furthermore, compared with code where the algorithms had been expanded manually to call the target functions directly, the performance was essentially the same. However, due to the extensive template instantiations, the compilation of code using the BL requires more resources than corresponding code that does not use the BL.

5 Conclusions

Using STL algorithms and function objects is a step towards adopting a functional programming style. However, only a subset of C++ function objects can be used with STL algorithms. Function object adaptors and binders in the STL provide some means to adapt functions for STL algorithms, but the solution is insufficient.

This article proposed a novel argument binding mechanism and a generic binder library based on this mechanism. Unbound arguments are specified directly in the argument lists as opposed to an index within the function name of the binder. This allows much greater flexibility in bind expressions. Arguments of nonmember functions, const and nonconst member functions, as well as STL function objects can be bound. The types and number of the arguments can be arbitrary.

The binding syntax is very simple and intuitive. In particular, no adaptors (cf. ptr_fun or mem_fun in STL) are required prior to binding. Moreover, the library supports recursive binding, which provides a versatile function composition mechanism.

The library is type safe. Type errors in bind invocations result in compile time errors. As a downside, these error messages are sometimes lengthy and difficult to interpret. The proposed binding mechanism does not induce any performance degradation, it is as efficient as the standard STL binding mechanism.

References

1. Stepanov, A. A., Lee, M.: The Standard Template Library. Hewlett-Packard Laboratories Technical Report HPL-94-34(R.1) (1994) www.hpl.hp.com/techreports.
2. International Standard, Programming Languages – C++. ISO/IEC:14882 (1998).
3. The SGI Standard Template Library. Silicon Graphics Computer Systems Inc. www.sgi.com/Technology/STL.
4. Liskov, B., Curtis, D., Day, M., Ghemawat S., Gruber R., Johnson, P., Myers A. C.: Theta Reference Manual, Preliminary version. Programming Methodology Group Memo 88 1995, MIT Lab. for Computer Science www.pmg.lcs.mit.edu/Theta.html.
5. Agents, iterators and introspection. Interactive Software Engineering Inc. Technology paper www.eiffel.com.
6. Stroustrup, B.: The C++ Programming Language - Third Edition. Addison-Wesley, Reading, Massachusetts 1997.
7. Järvi J.: Tuples and multiple return values in C++. submitted for publication, see TUCS Technical Report 249 (1999) www.tucs.fi/publications.
8. Veldhuizen, T.: Using C++ Template Metaprograms. C++ Report **7** (1995) 36–43.
9. Järvi J.: Compile Time Recursive Objects in C++. Proceedings of the TOOLS 27 conference (Beijing Sept. 1998) 66–77. IEEE Computer Society Press.
10. Czarnecki, K.: Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. Ph.D. Thesis, Technische Universitt Ilmenau, Germany 1998.
11. The GNU Compiler Collection. www.gnu.org/software/gcc/gcc.html.