Patterns in Datatype-Generic Programming

Jeremy Gibbons

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
`jeremy.gibbons@comlab.ox.ac.uk`

Abstract. *Generic programming* consists of increasing the expressiveness of programs by allowing a wider variety of kinds of parameter than is usual. The most popular instance of this scheme is the C++ Standard Template Library. *Datatype-generic programming* is another instance, in which the parameters take the form of datatypes. We argue that datatype-generic programming is sufficient to express essentially all the genericity found in the *Standard Template Library*, and to capture the abstractions motivating many *design patterns*. Moreover, datatype-generic programming is a precisely-defined notion with a rigorous mathematical foundation, in contrast to generic programming in general and the C++ template mechanism in particular, and thereby offers the prospect of better static checking and a greater ability to reason about generic programs. This paper describes work in progress.

1 Introduction

Generic programming [28, 19] is a matter of making programs more adaptable by making them more general. In particular, it consists of allowing a wider variety of entities as parameters than is available in more traditional programming languages.

The most popular instantiation of generic programming today is through the C++ Standard Template Library (STL). The STL is basically a collection of container classes and generic algorithms operating over those classes. The STL is, as the name suggests, implemented in terms of C++'s template mechanism, and thereby lies both its flexibility and its intractability.

Datatype-generic programming (DGP) is another instantiation of the idea of generic programming. DGP allows programs to be parameterized by a *datatype* or *type functor*. DGP stands and builds on the formal foundations of category theory and the *Algebra of Programming* movement [8, 7, 10], and the language technology of Generic Haskell [22, 12].

In this paper, we argue that DGP is sufficient to express essentially all the genericity found in the STL. In particular, we claim that various programming idioms that can at present only be expressed informally as *design patterns* [17] could be captured formally as datatype-generic programs. Moreover, because DGP is a precisely-defined notion with a rigorous mathematical foundation, in contrast to generic programming in general and the C++ template mechanism in particular, this observation offers the prospect of better static checking of and a greater ability to reason about generic programs than is possible with other approaches.

This paper describes work in progress — in fact, it describes work largely in the future. The United Kingdom's Engineering and Physical Sciences Research Council is funding a project called *Datatype Generic Programming*, starting around September 2003. The work described in this paper will constitute about a third of that project; a second strand, coordinated by Roland Backhouse at Nottingham, is looking at more of the underlying theory, including logical relations for modular specifications, higher-order naturality properties, and termination through well-foundedness; the remainder of the project consists of an integrative case study.

The rest of this paper is structured as follows. Section 2 describes the principles underlying the C++ Standard Template Library. Section 3 motivates and defines Datatype-Generic Programming, and explains how it differs from a number of similar approaches to genericity. Section 4 discusses the Design Patterns movement, and presents our case for the superiority of datatype genericity over informal prose for capturing patterns. Section 5 concludes by outlining our future plans for the DGP project.

2 Principles Underlying the STL

The STL [6] is structured around four underlying notions: *container types*, *iterators*, *algorithms*, and *function objects*. These notions are grouped into a hierarchy (in fact, a directed acyclic graph) of *concepts*, representing different abstractions and their relationships. The library is implemented using the C++ *template mechanism*, which is the only means of writing generic programs in C++. This section briefly analyzes these six principles, from a functional programmer's point of view.

2.1 The C++ Template Mechanism

The C++ template mechanism provides a means for classes and functions to be parametrized by types and (integral, enumerated or pointer) values. This allows the programmer to express certain kinds of abstraction that otherwise would not be available. A typical example of a function parametrized by a type is the function *swap* below:

```
template⟨class T⟩
void swap(T& a, T& b) {T c = a; a = b; b = c;}

main() {
  int i1 = 3, i2 = 4;              swap⟨int⟩(i1, i2);
  double d1 = 3.5, d2 = 4.5; swap⟨double⟩(d1, d2);
}
```

The same function template is instantiated at two different types to yield two different functions. Container classes form typical examples of parametrization of a class by a type; the example below shows the outline of a *Vector* class parametrized by size and by element type.

```
template⟨class T, int size⟩
class Vector {private : T values[size]; ...};

main() {
   Vector⟨int, 3⟩ v;
   Vector⟨Vector⟨double, 100⟩, 100⟩ matrix;
}
```

The same class template is instantiated three times, to yield a one-dimensional vector of three integers and a two-dimensional 100-by-100 matrix of doubles.

A template is to all intents and purposes a macro; little is or can be done with it until the parameters are instantiated, but the instantiations that this yields are normal code and can be checked, compiled and optimized in the usual way. In fact, the decision about which template instantiations are necessary can only be made when the complete program is available, namely at link time, and typically the linker has to call the compiler to generate the necessary instantiations.

The C++ template mechanism is really a *special-purpose, meta-programming* technique, rather than a general-purpose generic-programming technique. Meta-programming consists of writing programs in one language that generate or otherwise manipulate programs written in another language. The C++ template mechanism is a matter of meta-programming rather than programming because templated code is not actually 'real code' at all: it cannot be type-checked, compiled, or otherwise manipulated until the template parameter is instantiated. Some errors in templated code, such as syntax errors, can be caught before instantiation, but they are in the minority; static checking of templates is essentially impossible. Thus, a class template is not a formal construct with its own semantics — it is one of the ingredients from which such a formal entity can be constructed, but until the remaining ingredients are provided it is merely a textual macro. In a programming language that offers such a template mechanism as its only support for generic programming, there is no hope for a calculus of generic programs: at best there can be a calculus of their specific instances.

The template mechanism is a special-purpose, as opposed to general-purpose, meta-programming technique, because only limited kinds of compile-time computation can be performed. Actually, the mechanism provides surprising expressive power: Unruh [38] demonstrated the disquieting possibility of a program whose compilation yields the prime numbers as error messages, Czarnecki and Eisenecker [13] show the Turing-completeness of the template mechanism by implementing a rudimentary LISP interpreter as a template meta-program, and Alexandrescu [4] presents a tour-de-force of unexpected applications of templates. But even if technically template meta-programming has great expressiveness, it is pragmatically not a convenient tool for generating programs; applications of the technique feel like tricks rather than general principles. Everything computable is expressible, albeit sometimes in unnatural ways. A true general-purpose meta-programming language would support ‘programs as data’ as first-class citizens, and simple and obvious (as opposed to ‘surprising’) techniques for manipulating such programs [35].

There are several consequences of the fact that templated code is a meta-program rather than (a fragment of) a pure program. They all boil down to the fact that separate compilation of the templated code is essentially impossible; it isn’t real code until it is instantiated. Therefore:

– templated code must be distributed in source rather than binary form, which might be undesirable (for example, for intellectual property reasons);
– static error checking is in general precluded, and any errors are revealed only at instantiation time; moreover, error reports are typically verbose and unhelpful, because they relate to the consequences of a misuse rather than the misuse itself;
– there is a problem of ‘code bloat’, because different instantiations of the same templated code yield different units of binary code.

There is work being done to circumvent these problems by resorting to partial evaluation [39], but there is no immediate sign of a full resolution.

2.2 Container Types

A *container type* is a type of data structures whose purpose is to contain elements of another type, and to provide access to those elements. Examples include arrays, sequences, sets, associative mappings, and so on.

To a functional programmer, this looks like a *polymorphic datatype*; for example,

$$\mathbf{data}\ \mathit{List}\ \alpha = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)$$

A data structure of type $\mathit{List}\ \alpha$ for some α will indeed contain elements of type α, and will (through pattern-matching, for example) provide access to them. Such polymorphic datatypes can be given a formal semantics via the categorical notion of a *functor* [10], an operation simultaneously on types (taking a type α to the type $\mathit{List}\ \alpha$) and functions (taking a function of type $\alpha \to \beta$ to the map function of type $\mathit{List}\ \alpha \to \mathit{List}\ \beta$).

However, that response is a little too simple. Certainly, some polymorphic datatypes and some functors correspond to container types, but not all do. For example, consider the polymorphic type

$$\mathbf{data}\ \mathit{Transformer}\ \alpha = \mathit{Trans}\ (\alpha \to \alpha)$$

(The natural way to define this type in Haskell [34] is with a type synonym rather than a datatype declaration, but we’ve chosen the latter to make the point clearer.) There is no obvious sense in which a data structure of type $\mathit{Transformer}\ \alpha$ ‘contains’ elements of type α. Hoogendijk and de Moor [24] have shown that one wants to restrict attention to the functors with a *membership* operation. Technically, in their relational setting, the membership of a functor F is the largest lax natural transformation from F to Id, the identity functor; informally, membership is a non-deterministic mapping selecting an arbitrary element from a container data structure. Some functors, such as *Transformer*, have no membership operation, and so do not correspond to container types according to this definition.

2.3 Iterators

The essence of the STL is the notion of an *iterator*, which is essentially an abstraction of a pointer. The elements of a container data structure are made accessible by providing iterators over them; the container typically provides operations *begin*() and *end*() to yield pointers to the first element and to 'one step beyond' the last element.

Basic iterators may be compared for equality, dereferenced and incremented. But there are many different varieties of iterator: *input iterators* may be dereferenced only as R-values (for reading), and *output iterators* only as L-values (for writing); *forward iterators* may be deferenced in both ways, and may also be copied (so that multiple elements of a data structure may be accessed at once); *bidirectional iterators* may also be decremented; and *random-access iterators* allow amortized constant-time access to arbitrary elements.

Despite the name, iterators in the STL do not express exactly the same idea as the ITERATOR design pattern, although they have the same intent of 'providing a way to access the elements of an aggregate object sequentially without exposing its underlying representation' [17]. In fact, the proposed design in [17] is fairly close to an STL input iterator: an existing collection may be traversed from beginning to end, but the identities of the elements in the collection cannot be changed (although their state may be).

What all these varieties of iterator have in common, though, is that they point to individual elements of the data structure. This is inevitable given an imperative paradigm: as Austern [6] puts it, 'The moving finger writes, and having writ, moves on', and although under more refined iterator abstractions the moving finger may rewrite, and may move backwards as well as forwards, it is still a finger pointing at a single element of the data structure.

One functional analogue of iterators for traversing a data structure is the *map* operator that arises as the functorial action on element functions, acting on each element independently. More generally, one could point to *monadic maps* [15], which act on the elements one by one, using the monad to thread some 'state' through the computation.

However, lazy functional programmers are liberated by the availability of 'new kinds of glue' [26] for composing units of code, and have other options too. For example, they may use lists to achieve a similar separation of concerns: the interface between a collection data structure and its elements is via a list of these elements. The analogue to the distinction between input and output iterators (R-values and L-values) is the provision of one function to yield the *contents* of a data structure as a list of elements, and another to *generate* a new data structure from a given list of elements.

This functional insight reveals a rather serious omission in the STL approach, namely that it only allows the programmer to manipulate a data structure in terms of its elements. This is a very small window through which to view the data structure itself. A map ignores the shape of a data structure, manipulating the elements but leaving the shape unchanged; iterator-style access also (deliberately) ignores the shape, flattening it to a list. Neither is adequate for capturing problems that exploit the shape of the data, such as pretty-printers, structure editors, transformation engines and so on. A more general framework is obtained by providing *folds* to consume data structures and *unfolds* to generate them [18] — indeed, the *contents* and *generate* functions mentioned above are instances of folds and unfolds respectively, and a *map* is both a fold and an unfold.

2.4 Concepts

We noted in the previous section that the essence of the STL is a hierarchy of varieties of iterator. In the STL, the members of this hierarchy are called *concepts*. Roughly speaking, a concept is a set of requirements on a type (in terms of the operations that are available, the laws they satisfy, and the asymptotic complexities in time and space); equivalently, a concept can be thought of as the set of all types satisfying those requirements.

Concepts are not part of C++; they are merely an artifact of the STL. An STL reference manual [6] can do no more than to describe a concept in prose. Consequently, it is a matter of informal argument rather than formal reasoning whether a given type is or is not a model of a particular concept. This is a problem for users of the STL, because it is easy to make mistakes by using an inappropriate type in a particular context: the compiler cannot in general check the validity of a particular use, and tracking down errors can be tricky. There have been some valiant attempts to address this problem by programming idioms [36, 31] or static analysis [21], but ultimately the language seems to be a part of the problem here rather than a part of the solution.

The solution seems obvious to the Haskell programmer: use type classes [29]. A type class captures a set of requirements on a type, or equivalently it describes the set of types that satisfy those requirements. (Type classes are more than just interfaces: they can provide default implementations of operations too, and type class inference amounts to automatic selection of an implementation.) Type classes are only an approximation to the notion of a concept in the STL sense, because they can capture only the signatures of operations and not their extensional (laws) or intensional (complexity) semantics. However, they are statically checkable within the language, which is at least a step forwards: C++ concepts cannot even capture signatures formally. The Haskell collection class library Edison [11, 33] uses type classes formally in the same way that STL uses concepts informally.

2.5 Algorithms and Function Objects

The bulk of the STL, and indeed its whole raison d'être, is the family of generic *algorithms* over container types made possible by the notion of an iterator. These algorithms are general-purpose operations such as searching, sorting, comparing, copying, permuting, and so on. Iterators decouple the algorithms from the container types on which they operate: the algorithm is described in terms of an abstract iterator interface, and is then applicable to any container type on which an appropriate iterator is available.

There is no new insight provided by the algorithms per se; they arise as a natural consequence of the abstractions provided (whether informally as concepts or formally as type classes) to access the elements of container types. In the STL, algorithms are represented as function templates, parametrized by models of the appropriate iterator concept. To a Haskell programmer, algorithms in this sense correspond to functions with types qualified by a type class.

The remaining principle on which the STL is built is that of a *function object* (sometimes called a 'functor', but in a different sense that the functors of category theory). Function objects are used to encapsulate function parameters to algorithms; typical uses are for parametrizing a search function by a predicate indicating what to search for, or a sorting procedure by an ordering.

Function objects also yield no new insight to the functional programmer. In the STL, a function object is represented as an object with a single method which performs the function. This is essentially an instance of the STRATEGY design pattern [17]. To a functional programmer, of course, function objects are unnecessary: functions are first-class citizens of the language, and a function can be passed as a parameter directly.

3 Datatype Genericity

We propose a new paradigm for generic programming, which we have called *datatype-generic programming* (DGP). The essence of DGP is the parametrization of values (for example, of functions) by a *datatype*. We use the term 'datatype' here in the sense discussed in Section 2.2: a container type, or more formally a functor with a membership operation. For example, '*List*' is a datatype, whereas '*int*' is merely a type.

(Since a datatype is one type parametrized by another — 'lists of αs, for some type α' — and a datatype-generic program is a program parametrized in turn by such a type-parametrized type, we toyed briefly with the idea of describing our proposal as for a *'type-parametrized–type'—parametrized theory of programming*, or TPTPTP for short. But we decided that was a bit of a mouthful.)

3.1 An Example of DGP

Consider for example the parametrically polymorphic programs *maplist*,

$$\begin{aligned}
&\mathit{maplist} :: (\alpha \to \beta) \to \mathit{List}\ \alpha \to \mathit{List}\ \beta\\
&\mathit{maplist}\ f\ \mathit{Nil} &&= \mathit{Nil}\\
&\mathit{maplist}\ f\ (\mathit{Cons}\ a\ x) &&= \mathit{Cons}\ (f\ a)\ (\mathit{maplist}\ f\ x)
\end{aligned}$$

and (for the appropriate definition of the *Tree* datatype) *maptree*,

$$\begin{aligned}
&\mathit{maptree} :: (\alpha \to \beta) \to \mathit{Tree}\ \alpha \to \mathit{Tree}\ \beta\\
&\mathit{maptree}\ f\ (\mathit{Tip}\ a) &&= \mathit{Tip}\ (f\ a)\\
&\mathit{maptree}\ f\ (\mathit{Bin}\ x\ y) &&= \mathit{Bin}\ (\mathit{maptree}\ f\ x)\ (\mathit{maptree}\ f\ y)
\end{aligned}$$

Both of these programs are already quite generic, in the sense that a single piece of code captures many different specific instances. However, the two programs are themselves clearly related, and a DGP language would allow their common features to be captured in a single definition *map*:

$$\begin{aligned}
&\mathit{map}\langle \mathit{Unit}\rangle\ () &&= ()\\
&\mathit{map}\langle \mathit{Const}\ a\rangle\ x &&= x\\
&\mathit{map}\langle +\rangle\ f\ g\ (\mathit{Inl}\ u) &&= \mathit{Inl}\ (f\ u)\\
&\mathit{map}\langle +\rangle\ f\ g\ (\mathit{Inr}\ v) &&= \mathit{Inr}\ (g\ v)\\
&\mathit{map}\langle \times\rangle\ f\ g\ (u, v) &&= (f\ u, g\ v)
\end{aligned}$$

This single definition is parametrized by a datatype; in this case it is defined by structural induction over a grammar of datatypes. The two parametrically polymorphic programs are of course instances of this one datatype-generic program: $\mathit{maplist} = \mathit{map}\langle \mathit{List}\rangle$ and $\mathit{maptree} = \mathit{map}\langle \mathit{Tree}\rangle$.

At first glance, this looks rather like a generic algorithm that could have come from the STL, and indeed in this case that is a valid analogy to make: *map*-like operations can be expressed in the STL. However, the crucial difference is that DGP allows a program to *exploit* the shape of the data on which it operates. For example, one could write datatype-generic functions to encode a data structure as a bit string and to decode the bit string to regenerate the data structure [27]: the *shape* of the data structure is related to the *value* of the bitstring. A more sophisticated example involves Huet's 'Zipper' [25] for efficiently but purely functionally representing a tree with a cursor position; different types of tree require different types of zipper, and it is possible [1, 23] to write datatype-generic operations on the zipper: here, the shape of one data structure determines the shape of an auxilliary data structure in a rather complicated fashion. Neither of these examples are possible with the STL.

3.2 Isn't This Just. . . ?

As argued above, the parametrization of programs by datatypes is not the same as *generic programming* in the STL sense. The latter allows *abstraction from* the shape of data, but not *exploitation of* the shape of data. Indeed, this is why we chose a new term 'DGP' instead of simply using 'GP': we would prefer the latter term, but feel that it has already been appropriated for a more specific use than we would like. (For example, one often sees definitions such as 'Generic programming is a methodology for program design and implementation that separates data structures and algorithms through the use of abstract requirement specifications' [37, p19]. We feel that such definitions reduce generic programming to good old-fashioned abstraction.)

DGP is not the same thing as *meta-programming* in general, and template meta-programming in particular. Meta-programming is a matter of writing programs that generate or otherwise manipulate other programs. For example, C++ template meta-programs yield ordinary C++ code when instantiated (at least notionally, although the code so generated is typically never seen); they are not ordinary C++ programs in their own right. A meta-program for a given programming language is typically not a program written in that language, but one written in a meta-language that generates the object program when instantiated or executed. In contrast, a datatype-generic program is a program in its own right, written in (perhaps an enrichment of) the language of the object program.

Neither is DGP the same thing as polymorphism, in any technical sense we know. It is clearly not the same thing as ordinary *parametric polymorphism*, which allows one to write a single program that can manipulate both lists of integers and lists of characters, but does not allow one to write a single program that manipulates both lists of integers and trees of integers. We also believe (but have yet to study this in depth) that DGP is not the same thing as *higher-order parametric polymorphism* either, because in general the programs are not parametric in the functor parameter: if they were, they might manipulate the shape of data but could not compute with it, as with the encoding and decoding example cited above.

Nor is it the same thing as *dependently typed programming* [5], which is a matter of parametrizing types by values rather than values by types. Dependent types are very general and powerful, because they allow the types of values in the program to depend on other values computed by that program; but by the same token they rule out the possibility of most static checking. (A class template parametrized by a value rather than a type bears some resemblance to type dependent on a value, but in C++ the actual template parameters must be statically determined for instantiation at compile time, whereas dependent type theory requires no such separation of stages.) It would be interesting to try to develop a calculus of dependently typed programming, but that is a different project altogether, and a much harder one too.

Finally, DGP is not simply Generic Haskell [12], although the datatype-generic program for *map* we showed above is essentially a Generic Haskell program. The Generic Haskell project is concentrating on the design and implementation of a language that supports DGP, but is not directly addressing the problem of developing a calculus of such programs. Our project has strong connections with the Generic Haskell project, and we are looking forward to making contributions to the design based on our theory-driven insights, as the language is making contributions to the theory by posing the question of how it may be used. However, Generic Haskell is just one possible implementation technique for DGP.

4 Patterns of Software

A *design pattern* 'systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems' [17]. The intention is to capture best practice and experience in software design in order to facilitate the education of novices in what constitutes good designs, and the communication between experts about those good designs. The software patterns movement is based on the work of Christopher Alexander, who for over thirty years has been leading a similar movement in architecture [3, 2].

It could be argued that many of the patterns in [17] are idioms for mimicking DGP in languages that do not properly support such a feature. Because of the lack of proper language support, a pattern can generally do no better than to motivate, describe and exemplify an idiom: it can refer indirectly to the idiom, but not present the idiom directly as a formal construction. For example, the ITERATOR pattern shows how an algorithm that traverses the elements of a collection type can be decoupled from the collection itself, and so can work with new and unforeseen collection types; but for each such collection type an appropriate new ITERATOR class must be written. (The programmer may be assisted by the library, as in Java [20], or the language, as in C♯ [14], but still has to write something for each new collection type.) A language that supported DGP would allow the expression of a single datatype-generic program directly applicable to an arbitrary collection type: perhaps a function to yield the elements as a lazy list, or a *map* operation to transform each element of a collection.

The situation is no better with the STL than with design patterns. We argued above that iterators in the STL sense are more general than the ITERATOR pattern. Nevertheless, C++ provides no support for defining the iterator concept, so it too can only be referred to indirectly; and again, for every new collection type an appropriate implementation of the concept must be provided.

As another example, the VISITOR pattern [17] allows one to decouple a multivariant datatype (such as abstract syntax trees for a programming language) from the specific traversals to be performed over that datatype (such as type checking, pretty printing, and so on), allowing new traversals to be added without modifying and recompiling each of the datatype variants. However, each new datatype entails a new class of VISITOR, implemented according to the pattern. A DGP language would allow one to write a single datatype-generic traversal operator (such as a *fold*) once and for all multivariant datatypes.

(Alexandrescu [4] does present a 'nearly generic' definition of the VISITOR pattern using clever template meta-programming, but it relies on C++ macros, and still requires the foresight in designing the class hierarchy to insert a call to this macro in every class in the hierarchy that might be visited.)

It is sometimes said that patterns cannot be automated; anything that can be captured completely formally is too restricted to be a proper pattern. Alexander describes a pattern as giving 'the core of the solution to [a] problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice' [3]; Gamma et al. state that 'design patterns are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is' [17]. We are sympathetic to the desire to ensure that patternity does not become a synonym for 'a good idea', but do not feel that that means we should give up on attempts to formalize patterns.

Alexander, in his foreword to Gabriel's book [16], hopes that the software patterns movement will yield 'programs which make you gasp because of their beauty'. We think that's a goal worth aiming for, however optimistically. We have yet to see a meta-programming framework that supports beautiful programming (although we confess to being impressed by the intricate possibilities of template meta-programming demonstrated by [4]), but we have high hopes that datatype-generic programs could be breathtakingly beautiful.

5 Future Plans

The DGP project is due to start around September 2003; the work outlined in this paper constitutes about a third of the total. One of the initial aims of this strand will be an investigation into the relationships between generic programming (as exhibited in libraries like the STL), structural and behavioural design patterns (as described by [17]), and the mathematics of program construction (epitomized by Hoogendijk and de Moor's categorical characterization of datatypes [24]).

In the short term, we intend to use the insights gained from this investigation to prototype a datatype-generic collection library in Generic Haskell [12] (perhaps as a refinement of Okasaki's Edison library [33]). This will allow us to replace type-unsafe meta-programming with type-safe and statically checkable datatype-generic programming. Ultimately, however, we hope to be able to apply these insights to programming in more traditional object-oriented languages, perhaps by compilation from a dedicated DGP language.

But the real purpose of the project will be to generalize theories of program calculation such as Bird and de Moor's relational 'algebra of programming' [10], to make it more applicable to deriving the kinds of programs that users of the STL write. This will link with Backhouse's strand of the DGP project, which is looking at more theoretical aspects of datatype genericity: higher-order naturality properties, logical relations, and so on. We intend to build on this work to develop a calculus for generic programming.

More tangentially, we have been intrigued by similarities between some of the more esoteric techniques for template meta-programming [13, 4] and some surprising possibilities for computing with type classes in Haskell [32, 30, 9]. It isn't clear yet whether those similarities are a coincidence or evidence of some deeper correspondence; in the light of our arguments in this paper that type classes are the Haskell analogue of STL concepts, we suspect there may be some deep connection here.

6 Acknowledgements

The help of the following people and organizations is gratefully acknowledged:

– Roland Backhouse, Graham Hutton, Ralf Hinze and Johan Jeuring, for their contributions to the DGP grant proposal;
– Richard Bird, for inspiring and encouraging this line of enquiry;
– Tim Sheard, for his elegant definition of generic programming;
– EPSRC grant GR/S27078/01, for financial support.

References

1. Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of containers. In Martin Hofmann, editor, *LNCS 2701: Typed Lambda Calculi and Applications*, pages 16–30. Springer-Verlag, 2003.
2. Christopher Alexander. *The Nature of Order.* Oxford University Press, To appear in 2003.
3. Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language.* Oxford University Press, 1977.
4. Andrei Alexandrescu. *Modern C++ Design.* Addison-Wesley, 2001.
5. Lennart Augustsson. Cayenne: A language with dependent types. *SIGPLAN Notices*, 34(1):239–250, 1999.
6. Matthew H. Austern. *Generic Programming and the STL.* Addison-Wesley, 1999.
7. R.C. Backhouse, P. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J. van der Woude. Polynomial relators. In M. Nivat, C.S. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of the 2nd Conference on Algebraic Methodology and Software Technology, AMAST'91*, pages 303–326. Springer-Verlag, Workshops in Computing, 1992.
8. R.C. Backhouse, P. de Bruin, G. Malcolm, T.S. Voermans, and J. van der Woude. Relational catamorphisms. In Bernhard Möller, editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications*, pages 287–318. Elsevier Science Publishers B.V., 1991.
9. Roland Backhouse and Jeremy Gibbons. Programming with type classes. Presentation at WG2.1#55, Bolivia, January 2001.
10. Richard S. Bird and Oege de Moor. *Algebra of Programming.* Prentice Hall, 1997.
11. Andrew Bromage. Haskell Foundation Library. `www.sourceforge.net/projects/hfl/`, 2002.
12. Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Universiteit Utrecht, 2001.
13. Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools and Applications.* Addison-Wesley, 2000.
14. Peter Drayton, Ben Albahari, and Ted Neward. *C♯ in a Nutshell.* O'Reilly, 2002.
15. Maarten Fokkinga. Monadic maps and folds for arbitrary datatypes. Dept INF, Univ Twente, June 1994.
16. Richard P. Gabriel. *Patterns of Software: Tales from the Software Community.* Oxford University Press, 1996.
17. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
18. Jeremy Gibbons. Origami programming. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming.* Palgrave, 2003.
19. Jeremy Gibbons and Johan Jeuring, editors. *Generic Programming.* Kluwer Academic Publishers, 2003.

20. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Mass., 2000.
21. Douglas Gregor and Sybille Schupp. Making the usage of STL safe. In Gibbons and Jeuring [19].
22. Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43:129–159, 2002. Earlier version appears in LNCS 1837: Mathematics of Program Construction, 2000.
23. Ralf Hinze and Johan Jeuring. Weaving a web. *Journal of Functional Programming*, 11(6):681–689, 2001.
24. Paul Hoogendijk and Oege de Moor. Container types categorically. *Journal of Functional Programming*, 10(2):191–225, 2000.
25. Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.
26. John Hughes. Why functional programming matters. *Computer Journal*, 1989.
27. Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–72, 2002.
28. Mehdi Jazayeri, Rüdiger G. K. Loos, and David R. Musser, editors. *Generic Programming.* Springer-Verlag, 2000.
29. Mark P. Jones. *Qualified Types: Theory and Practice.* DPhil thesis, University of Oxford, 1992.
30. Conor McBride. Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming*, 12(4&5):375–392, 2002.
31. Brian McNamara and Yannis Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming*, October 2000.
32. Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. Functional logic overloading. In *Symposium on Principles of Programming Languages*, pages 233–244, 2002.
33. Chris Okasaki. An overview of Edison. Haskell Workshop, 2000.
34. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.
35. Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell Workshop*, 2002.
36. Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming*, October 2000.
37. Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library.* Addison-Wesley, 2002.
38. Erwin Unruh. Prime number computation. ANSI X3J16-94-0075/ISO WG21-462, 1994.
39. Todd Veldhuizen. Five compilation models for C++ templates. In *First Workshop on C++ Template Programming*, October 2000.