Generic and Indexed Programming (Project Paper)

Jeremy Gibbons, Meng Wang, and Bruno C. d. S. Oliveira

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
{jg,menw,bruno}@comlab.ox.ac.uk

Abstract

The EPSRC-funded *Generic and Indexed Programming* project will explore the interaction between *datatype-generic programming* (DGP) — programs parametrized by the shape of their data — and *indexed programming* (IP) — lightweight dependently-typed programming, with programs indexed by type-level representations of properties. Integrating these two notions will provide new ways for programmers to *capture abstractions*.

The project builds on insights from our recent work in DGP, which has investigated both *programming techniques* (including reasoning about generic programs, and using them to capture design patterns precisely), and *language mechanisms* (particularly lightweight approaches: patterns for simulating highly-expressive techniques in familiar but apparently less-expressive languages). Firstly, these lightweight techniques, which we have been embedding in Haskell's still relatively expressive type system, are in fact applicable to even less expressive but more *popular mainstream languages* such as Java and C#. Secondly, the techniques are more applicable than we first thought; in particular, they offer a solution to the so-called 'expression problem': safe combination of independent extensions along multiple dimensions. Thirdly, there is a synergy between DGP and IP: DGP makes IP more appealing, because the effort of stating properties can be amortized over more programs; IP provides a mechanism for DGP, because the indices can be representations of data's shape.

1 INTRODUCTION

The essence of computer science is *abstraction*: identifying patterns in code, and expressing these patterns in machine-readable forms. But capturing these patterns is often a challenge; programmers are continually struggling with their tools. There is often a gulf between what they know and what their language lets them state directly; knowledge that should be abstracted and analyzable is instead dispersed and unreachable. Languages evolve by allowing the programmer to say more about their programs, in a format that compilers and another language tools can exploit.

The *Generic and Indexed Programming* (GIP) project at Oxford plans to develop mechanisms allowing programmers to state *properties of their programs*, and to validate these properties. Specifically, properties are expressed as types, and property validation is type checking. The kinds of property we have in mind are *indexes* (such as size, shape or state), and the representation is as *generalized algebraic datatypes*.

2 GENERIC PROGRAMMING

Generic programming is about making programming languages more flexible without compromising safety. Both sides of this equation are important, and becoming more so as we seek to do more and more with computer systems, while becoming ever more dependent on their reliability.

Generic programming usually manifests itself as a kind of parametrization. By abstracting from the differences in what would otherwise be separate but similar specific programs, one can make a single unified generic program. Instantiating the parameter in various ways retrieves the various specific programs, and ideally some new ones too. The different interpretations of the term 'generic programming' arise from different notions of what constitutes a 'parameter'.

The term has a long history, and a corresponding variety of interpretations. To some people, it means parametric polymorphism; to some, it means libraries of algorithms and data structures; to some, it means reflection and meta-programming; to some, it means polytypism. We favour the latter interpretation, but to avoid confusion, for our recent work in the area we coined the more specific term *datatype-generic programming* (DGP).

We are interested particularly in programs parametrized by datatypes, that is, by type constructors such as 'list of' and 'tree of'. Typical examples are pretty printers and marshallers, which can be defined once and for all for lists, trees, and so on, in a typesafe way. This is not just parametric polymorphism, which is abstraction from the 'integer' rather than the 'list' in 'lists of integers'; it is not just about libraries of algorithms and data structures as in the C++ Standard Template Library, which is really just an outworking of parametric polymorphism (together with some ad-hoc polymorphism in the *concepts* that cannot — yet [10] — be formally stated but that parameters are required to model); and it is ideally not just reflection and meta-programming, which are typically dynamic and not typesafe.

Preceding work on languages to support datatype-generic programming focused mostly on special-purpose languages for supporting a particular view of polytypism. For example, *PolyP* [16] programs are parametrized by regular functors, and *Generic Haskell* (GH) [13] programs by sums of products. Lämmel and Peyton Jones' *Scrap your Boilerplate* (SyB) technique [18] takes a different approach, relying on a type-safe cast operator and a small number of generic combinators, and providing nominal rather than structural polytypism.

More recently, some *lightweight* approaches to generic programming have begun to emerge. Hinze and Cheney [4] were among the first to propose a technique requiring only Haskell 98 plus existential types, a very mild extension supported by most Haskell compilers. Hinze and Cheney's work evolved into what they called *first-class phantom types* [11]: type parameters that are used to record properties of data rather than elements of data structures. Similar ideas have arisen in a number of neighbouring fields, under the names *guarded recursive datatypes* [42], *indexed type families* [44], *inductive families* [6], and *equality-qualified types* [32], among others. These developments inspired the recent *generalized algebraic datatype*

(GADT) [30] extension to the Glasgow Haskell Compiler (GHC), which we discuss further in Section 3.

The lightweight approaches to generic programming have shown that existing features of current programming languages — in particular, type classes, existential types and generalized algebraic datatypes — can be used to build libraries for generic programming. Special-purpose languages or extensions are not needed; these general-purpose techniques are sufficient. This is very appealing; fewer specialized tools means less chance for fragmentation in the various programming language communities: theorists, implementors, and programmers.

3 INDEXED PROGRAMMING

Programming languages are progressively allowing the programmer to express more precise properties of their programs, in a way that compilers can exploit for safety and for efficiency. In particular, recent developments allow the programmer to specify properties about the *shape of data* (such as the dimensions of a matrix, or the balancing of a tree) and the *state of components* (such as safety or security properties of an agent in a protocol), in terms of richer type systems.

Developments of this kind include *proof-carrying code* [26], Programatica's *certificates* of validation [37], *nested datatypes* [2], *indexed types* [43], and particularly *generalized algebraic datatypes* [30]. What they all have in common is to lift properties of programs that would otherwise be available only dynamically, if at all, and make them statically checkable and analysable. In a sense, they are lightweight approaches to dependently-typed programming, occupying a sweet spot reaping some of the benefits without incurring all of the costs (for example, in difficulty of type-checking, and accessibility of programming). Even straightforward test-driven development, as embodied in tools such as QuickCheck and JUnit, encourages the machine-readable statement of code properties.

GADTs allow the packing of syntactic type equality constraints into individual data constructors. Pattern matching can exploit these type constraints, allowing more refined type judgements. The following example shows a GADT guaranteeing well-typing of well-formed expressions.

```
data Exp a where
  Zero :: Exp Int
  Succ :: Exp Int → Exp Int
  Pair :: Exp b → Exp c → Exp (b,c)
```

In contrast to normal algebraic data types, the result type of each constructor of *Exp* is refined: for example, constructor *Zero* yields *Exp Int* rather than *Exp a*. An evaluator for *Exp* can be defined as follows.

```
eval :: Exp a → a
eval Zero       = 0
eval (Succ e)   = 1 + eval e
eval (Pair x y) = (eval x, eval y)
```

Note that in the first clause, pattern *Zero* has type *Exp Int*; matching this against the required argument type *Exp a* induces a constraint $a = Int$. Under this constraint, the type *Int* of the right-hand side matches the required type *a*. The other clauses are type-checked similarly.

This feature opens up a brave new world of programming with properties. However, we are only just starting to explore this new world, and currently trying to do so using old-world tools. GHC now supports GADTs [30], but they are not in the Haskell 98 standard or any other popular implementation of the language; they are currently under consideration for inclusion in Haskell$'$, the next version of the Haskell standard. There are promising signs that GADTs may follow generics into mainstream languages such as Java and C# [17]. Now is the time to explore the design space, and see what works and what does not. That is the goal of the GIP project.

4 APPLICATIONS OF INDEXED PROGRAMMING

Something like the 'typed expressions' example in Section 3 is used for motivation in most papers about GADTs, but IP is much more widely applicable than this. To give some idea of the breadth of applications, we present a representative selection here. Most of these are already well-known in the literature, but the Mini Nim example in Section 4.4 is new.

A more comprehensive catalogue of IP applications is presented in Appendix A. Some of the examples can be implemented in a lightweight manner in existing languages, such as Haskell with GADTs; others seem to require heavier language machinery. Expanding and investigating this catalogue will be our first strand of work on the GIP project, as discussed in Section 5.1; this will provide us with a benchmark suite on which to validate the other results of the project.

4.1 Enumerations

The simplest class of index is an enumeration. For example, *red-black trees* are binary trees in which (among other constraints) nodes are labelled with a colour, red or black, and red nodes have black children. That property can be captured by indexing by colour, and constraining the tree constructors, very like the typed expression example above.

```
data R
data B
data RBTree a c where
  Empty ::                              RBTree a B
  Red   :: RBTree a B → a → RBTree a B  → RBTree a R
  Black :: RBTree a c → a → RBTree a c' → RBTree a B
```

(This presentation omits the height constraint, which uses numeric indexes as described in Section 4.2 below.)

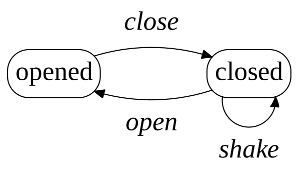
As a more interesting example, transitions in a state graph can be indexed by their end states, enforcing safety properties in a protocol. For example, consider the *ketchup problem*: a bottle may be open or closed, but it is safe to shake the bottle only in its closed state.

```
data O
data C

data Edge s1 s2 where
  Open  :: Edge O C
  Close :: Edge C O
  Shake :: Edge C C

data Path s1 s2 where
  Empty :: Path s s
  PCons :: Edge s t → Path t u → Path s u

scenario :: Path O O
scenario = PCons Open (PCons Shake (PCons Close Empty))
```



Here, the datatypes *O* and *C* represent the states 'open' and 'closed'. The type *Edge* of state transitions is indexed by states, and a *Path* is well-typed if and only if its sequence of transitions is safe. This example is frivolous, but the principle applies to more serious problems, such as resource locking.

4.2 Numbers

Natural numbers are probably the most widely used index. Many important properties of datatypes, such as the black-height of a red-black tree or the length of a vector, can be captured by natural numbers.

```
data Z
data S n

data Vector a n where
  VNil  ::                     Vector a Z
  VCons :: a → Vector a n → Vector a (S n)
```

Natural numbers are encoded here at the type level; for example, the number 3 is represented by the type $S\ (S\ (S\ Z))$. The datatype *Vector a n* is parametrized by its element type *a* and indexed by its size *n*. Now we can express the fact that zipping two vectors of the same length yields a result also of that length, and make it a type error to attempt to zip vectors of different lengths.

```
vzip :: Vector a n → Vector b n     → Vector (a,b) n
vzip   VNil          VNil           = VNil
vzip   (VCons x xs)  (VCons y ys)   = VCons (x,y) (vzip xs ys)
```

4.3 Types

Type representations serving as directives to datatype-generic functions have been well-studied in the literature [4, 11]. For example, consider a little language of sum-of-product datatypes, built from integers with the following operations:

```
data Unit     = Unit
data Sum a b  = Inl a | Inr b
data Prod a b = P a b
```

Members of this family can be represented by terms of type *Rep*; but by using indexing, the value of the representation can be reflected at the type level — the type *t* is represented by a (in fact, the unique total) term of type *Rep t*.

```
data Rep a where
  RI  :: Rep Int
  RU  :: Rep Unit
  RS  :: Rep a → Rep b → Rep (Sum a b)
  RP  :: Rep a → Rep b → Rep (Prod a b)
```

Now a datatype-generic equality function can traverse the type representation to direct the comparison, without sacrificing type safety.

```
rEq                 :: Rep t → t → t → Bool
rEq RI t₁ t₂         = t₁ == t₂
rEq RU t₁ t₂         = case (t₁, t₂) of
                         (Unit, Unit)      → True
rEq (RS ra rb) t₁ t₂ = case (t₁, t₂) of
                         (Inl a₁, Inl a₂)  → rEq ra a₁ a₂
                         (Inr b₁, Inr b₂)  → rEq rb b₁ b₂
                         _                 → False
rEq (RP ra rb) t₁ t₂ = case (t₁, t₂) of
                         (P a₁ b₁, P a₂ b₂) → rEq ra a₁ a₂ ∧ rEq rb b₁ b₂
```

4.4 Data structures

Indexing by natural numbers or by representations of an infinite family of types involve recursive data structures at the type level. Another recurring example of recursive indices involves indexing a value with a proof of some property of that value. For example, a recent proposal [1] showed how to capture precisely the lambda terms normalizable under call-by-value, via indexing with the corresponding sequence of reductions. Space restrictions preclude the presentation of that example, but instead here is a simpler example of indexing by a proof witness.

The game of *Mini Nim* involves a pile of matchsticks. Players take turns to remove either one matchstick or two, and the player who removes the last matchstick wins. The type *Position n r* is indexed by the number *n* of matchsticks, and the optimal result *r* for the next player. Clearly, being faced with an empty pile, you lose. If you have a pile with more matchsticks, and can turn it into a losing pile by taking one matchstick or two, you win; and conversely, if taking one matchstick or two both yield winning positions, you lose.

```
data Win
data Lose

data Position n r where
  Empty :: Position Z Lose
  Take1 :: Position n Lose                              → Position (S n) Win
  Take2 :: Position n Lose                              → Position (S (S n)) Win
  Fail  :: Position n Win → Position (S n) Win → Position (S (S n)) Lose
```

Note that the proof witness in this case is in fact not a linear sequence, as in the terminating lambda terms example, but rather tree-shaped, as most proofs are. Of course, a matchstick game is another rather frivolous example; but such proof-carrying code has many more serious applications [26].

Since proof terms are used as witnesses, it is important to guarantee their validity. Haskell's non-strict semantics does not enforce this, when a proof term is not dynamically inspected. For example, we can easily construct a bogus proof that an empty pile of matchsticks is a winning position:

```
bogus :: Position Z Win
bogus = ⊥
```

Perhaps a strict semantics, as Sheard has in Ωmega [32, 33], is more appropriate for computing with witnesses; it remains unclear how indexed programming can be successfully integrated with Haskell's lazy evaluation.

5 PROJECT STRANDS

This section outlines the six strands of work that make up the GIP project.

5.1 Capturing properties

Our previous work on the *Datatype-Generic Programming* project has led us to the realization that GADTs are a very useful tool in the generic programmer's toolkit. In particular, they allow the expression and static checking of otherwise-inexpressible properties of programs, such as the shape of data and the state of components. But they are a very recent innovation, and as yet are not well-supported by programming languages or well-understood by programmers. The GIP project will investigate these mechanisms for lightweight indexing of types by values. We will start by carrying out some initial case studies on programming with properties.

Although GADTs are a neat trick, programming with them currently can be quite hard work. Essentially, one has to resort to proving theorems in the language of types, assisting the type checker in verifying the stated properties. For example, as we saw in Section 4.2, it is a simple matter using GADTs to represent a type of vectors indexed by their length, and to express the constraint on 'zip' that its two arguments and its result should all have the same length. However, to express the constraint on indexing that the position should be less than the length requires a proof of this inequality; and to express precisely the shape of the ‘triangular’ vector of vectors returned by a ‘prefixes’ function requires a new type of length-indexed vectors with position-indexed elements.

In this strand of the project, we will carry out a number of case studies in programming with GADTs, expanding on those presented in Section 4. In doing so, we aim to compare the ergonomics of programming using GADTs with other competing approaches, such as nested datatypes [2], Dependent ML [43], Ωmega [32, 33], and Epigram [21]. We will use this experience to inform our development of notations for indexed programming.

One of the questions of notation that we wish to explore is the relationship between GHC’s current ‘untyped’ (or rather, *unkinded*) approach to type indices, and the *extensible kinds* in Sheard’s Ωmega. With the former, using Haskell’s syntax for kind declarations, length-indexed vectors are expressed as follows:

```
data Z
data S n
data Vector :: * → * → * where
  VNil  ::                    Vector a Z
  VCons :: a → Vector a n → Vector a (S n)
```

Here, the invariant that the second type parameter of *Vector* must be a type-level natural is not expressed explicitly, but is implicit in the fact that there is no way to construct a *Vector* of any other type. In contrast, extensible kinds would allow one to declare the kind of that second parameter:

```
kind Nat = Z | S Nat
data Vector :: * → Nat → * where
  VNil  ::                    Vector a Z
  VCons :: a → Vector a n → Vector a (S n)
```

This introduces a new closed kind *Nat* with two members, a type *Z* and a type constructor *S*, and mandates their use in the second type parameter of *Vector*. How important is this extra expressivity? That is, how likely and how costly are kind errors without it, and how much trouble is it to introduce it? Only time and case studies can help us answer this question.

5.2 Generics for GADTs

Most approaches to DGP assume a view of datatypes; for example, Generic Haskell works for sum-of-product algebraic datatypes. However, generalizing to GADTs departs from this framework; generic programming on GADTs requires some alternative view of datatypes. This issue is important, because programming with GADTs is substantially trickier than programming with regular ADTs, and generic programming is consequently even more attractive a proposition.

In recent work [15], Hinze, Löh and Oliveira demonstrated that the Scrap your Boilerplate approach to generic programming can be implemented using the so-called *spine view* of datatypes. Under this view, generic functions can be defined by induction over the structure of the spine, in much the same way that generic functions in GH can be defined by induction over the sum-of-products structure. One disadvantage of the SyB approach is that generic functions are limited to *generic consumers*, such as pretty printers. However, an advantage of SyB over GH is that it can also handle GADTs.

Expanding on this work, Hinze and Löh show [14] that *generic producers* such as parsers can be defined using the so-called *type-spine view*. They also propose a *lifted-spine view* for capturing datatype-generic functions such us *generic map*, which cannot be defined using unlifted spines. However, this approach requires two more views, and two more type representations. Furthermore, it is limited to unary type constructors; supporting type constructors of different arities would involve yet more views and representations, and further duplication.

Recently, we have been exploring McBride and Paterson's work [22] on applicative functors, a generalization of monads. We have observed that applicative functors are closely connected with the spine view. We conjecture that applicative functors generalize spine views, and so can be used to define generic functions on GADTs. One of the benefits of the extra generality is that applicative functors naturally support datatype-generic functions such as *map*. Unlike Hinze and Löh's proposal, this means that we do not incur any duplication of structure, or even extensions to current compilers. Furthermore, applicative functors have a solid theoretical foundation in terms of so-called *strong lax monoidal functors*, in contrast to the somewhat ad-hoc nature of SyB, and we expect that this will provide a cleaner algebra for reasoning about generic programs [31].

We are already collecting some results from this line of research: a recent paper [9] uses applicative functors to model and reason about the ITERATOR design pattern. Of course, generic iterators are just one application of generic programming, but this is just a small start.

5.3 Extensible generic functions

One view of generic programming is as defining type-indexed functions. The advantage of *structural* approaches to generic programming, such as the GH sum-of-products approach, is that it allows generic functions to be defined once for all types, even those yet to be conceived.

This works nicely in most situations, but not all. For example, a library for ordered sets might be built on top of a datatype of balanced binary trees. Generic programming allows the set library to exploit generic definitions of operations such as pretty printing and equality. But these generic definitions are not appropriate in this specific case, since both operations should in fact ignore the tree structure. We need some means of overriding generic behaviour without endangering modularity; this is not possible in most of the current approaches to generic programming.

In contrast, *nominal* approaches to generic programming, such as Haskell's type classes, stipulate a separate implementation of a type-indexed function for each type index of interest; structurally similar but nominally different types have unrelated implementations. This is more flexible — customized pretty-printing and equality functions can easily be provided for sets — but less reusable: it throws the baby of genericity out with the bathwater of inflexibility.

The challenge we plan to address in this strand is support for *extensible generic functions*, combining the benefits of the structural and nominal approaches. Generic functions are given default definitions following the structure of the type parameter, but this default can be overridden locally for specific named types. Extensible generic functions can be seen as a variant of Wadler's *expression problem* [41, 38] — the problem of supporting simultaneous independent extension of datatypes and functions. The ability to support open datatypes and open functions is the key; but in contrast to Hinze and Löh's approach [19], we aim to preserve static safety: applying a function to a variant for which it is not defined is a static type error. Without such extensibility, generic programming is actually very limiting in practical use. This is one of the problems with the *Generics for the Masses* (GM) [12] and *SyB Reloaded* [15] approaches.

We believe we know how to remove these limitations. Inspired by two datatype encodings in the lambda calculus, the *Church encoding* [3] and the *Parigot encoding* [29], Hinze's GM approach [12] uses a type class *Generic*, with the sum-of-products encoding of datatypes as class methods. An instance of *Generic* defines a generic function, by giving cases for sums, products and so on. Another optional class can be used as a dispatcher to infer the right instantiation of the generic function automatically [27]. However, because datatype encodings are class methods, this approach shares the limitation of non-extensibility of generic programming.

The *Generics as a Library* pattern [28] presents a variation of the GM technique that allows the dispatching class to take instantiations from classes other than *Generic*. Thus, extensions defined in additional classes can be used to define new cases with user-specified behaviour. Viewed from the perspective of a datatype encoding, this pattern can be seen as a form of open datatypes with open functions, thereby solving the problem of extensibility.

Just as functional languages provide datatypes directly, rather than forcing the programmer to use encodings, we will explore the possibility of adding primitive language support for this pattern. This would allow the user to use the pattern in an intuitive way, rather than a having to understand an intricate encoding.

The first issue that needs to be addressed is the integration of the extra language construct into a Haskell-like language. We will investigate the possibility of having evidence translation, similar to that of type classes, enabling type erasure. An approach similar to type classes' also gives us static safety and good support for separate compilation. We also expect to lift some of the limitations noticed in [28]: only top-level pattern matching; no dispatching on multiple type arguments; awkward encoding of mutually recursive functions. The use of a Parigot encoding instead of a Church encoding may help; there are also some interesting ideas in EML [25] and λL [39] worth exploring.

5.4 Design patterns as a library

Recent versions of Java and C# provide support for parametric polymorphism. When combined with object-oriented subtyping (what Cardelli and Wegner call *inclusion polymorphism*), this makes the type systems of these two languages remarkably powerful. It turns out that this expressive power is sufficient to implement lightweight encodings of DGP in those languages.

Kennedy and Russo [17] showed that a mild extension to the type constraint mechanisms of these recent versions of Java and C# allows a form of GADTs to be encoded. They presented a variety of compelling examples, such as strongly typed evaluators, typed LR parsing and typed representations. Even without their extension, it is possible to encode some of the GADT programs in a type-safe manner; all the others can be encoded by resorting to casting.

In this strand, we will apply our recent work on lightweight encodings of generic programming [27, 28, 15] to mainstream object-oriented languages such as Java and C#, and experimental ones such as O'Caml and Scala, and integrate this with our work [7, 8, 9] on capturing design patterns as higher-order datatype-generic programs.

5.5 Type classes and GADTs

Lightweight indexed programming, as provided by GADTs and Sheard's Ωmega, can be seen as 'datatypes plus constraints'. Haskell's type classes, originally introduced to make ad-hoc polymorphism more disciplined [40], are also widely used as a way of imposing constraints on types. Despite some notable similarities, GADTs and type classes are mostly studied in isolation.

The design of Ωmega does not support type classes but, inspired by them, it contains a notion of propositions for implicitly propagating constraints using the type system [33]. These constraints usually represent proofs of properties, but they are generally boilerplate code because they only serve as witnesses. Since the proof objects are elements of *singleton types*, a special case of GADTs, they can be inferred automatically from the types. For example, we could encode a proposition for proofs that one natural number is at most another in Ωmega as follows:

$$\begin{array}{lll} \multicolumn{3}{l}{\textbf{prop}\ \mathit{LE} :: \mathit{Nat} \rightarrow \mathit{Nat} \rightarrow * \ \textbf{where}} \\ \quad \mathit{Base} & :: & \mathit{LE}\ Z\ n \\ \quad \mathit{Step} & :: & \mathit{LE}\ n\ m \rightarrow \mathit{LE}\ (S\ n)\ (S\ m) \end{array}$$

This proposition can be used to create a pair in which the first element is at most the second.

$$\begin{array}{l} \mathit{lePair} :: \mathit{LE}\ a\ b \Rightarrow a \rightarrow b \rightarrow (a,b) \\ \mathit{lePair}\ x\ y = (x,y) \end{array}$$

In this case, the proof is propagated implicitly. This can be a big help in managing invariants, since type inference will enforce the constraints. Furthermore, it becomes easier to erase proofs at runtime.

GADTs and type classes are closely related: they allow forms of closed and open type-indexed functions, respectively. Moreover, propositions (which are just special cases of GADTs) share much of the infrastructure of type classes. Interestingly, in current implementations of GHC, type classes and GADTs have quite orthogonal implementations, and there are even some problems of integration (although Sulzmann and Peyton Jones's recent work on System F_C [35, 36] may help with this).

We believe that the investigation of GADTs and type classes together has considerable merits. In this strand, we will investigate the combination from three angles. Firstly, neither GADTs nor type classes subsumes the other in term of expressiveness. However, sometimes it is possible to encode one with the other. We will study the advantages and disadvantages of both approaches, and capture the results as a pattern language for programming with witnesses.

Secondly, the combined power of GADTs and type classes may produce superior programs to those available using just one of these techniques in isolation. When programming with GADTs to capture shape invariants, a witness which specifies some property may be passed as an additional parameter to a function. This is very similar to the case of type representations in generic programming. It is well known that type classes can serve as dispatchers of type representations [27]. We will investigate to what extent type classes can serve as dispatchers for witnesses more generally.

Thirdly, both GADTs and type classes specify relations among types. The *functional dependency* type refinement ability of type classes which refines the relations into functions has also proven to be useful in stating shape invariants. We will investigate whether GADTs can be provided with similar extensions.

5.6 Impedence transformers

Generic metaprogramming frameworks like DrIFT [23] suffer from an impedence mismatch, because they handle untyped and unstructured (for example, purely textual) representations of object programs. It is difficult to statically check the metaprogram, once and for all, in particular to guarantee properties of the generated object programs. If it is a staged metaprogram, one can run the first stage and statically check the object program generated, before executing it in the second stage; but one must do this again and again for each run of the metaprogram. If the metaprogram is unstaged (as for example is the TEX macro language), it is not possible to obtain the intermediate object program at all, and one must resort to dynamically checking it at run-time.

Some of these problems are alleviated if the metaprogramming stage uses some abstract syntactic rather than purely textual representation (as in Template Haskell [34], for example), since it is then impossible to generate a syntactically incorrect object program; but still, the abstract syntax typically cannot capture precisely the type constraints of the object program, and so the problems of syntactically correct but ill-typed object programs persist.

Sheard [32] calls this a *semantic gap*, between the properties that the programmer knows about the object program to be generated, and those that the language lets them state. GADTs allow more of those properties to be expressed and statically checked, and hence work towards narrowing the semantic gap.

Similar issues of impedence mismatch arise in the kinds of multi-tier programming required for current enterprise application architectures. These typically involve something like HTML or JavaScript for the presentation layer, Java or C# for the logic layer, and SQL or XQuery for the data layer. Current approaches mandate different languages for the different layers, and no way of statically guaranteeing that (for example) the data entered into a form or retrieved from a database is of the type expected by the business logic. This causes significant problems for enterprise applications [20], and it would be an important contribution to do something about it.

Wadler [5] is working on *Links*, a single wide-spectrum language supporting distribution of appropriate tasks to the different layers by translation; the impedence mismatch between different notations is avoided by reference to a single integrated notation. Meijer [24] and others at Microsoft are working on *LINQ*, a set of extensions to the .Net framework with the same goal; but rather than introducing a new wide-spectrum language, they are extending C#, the language for the logic layer, with constructs to match the requirements of the presentation and data layers.

We conjecture that the kind of properties expressible with GADTs are sufficient to capture the additional information required to remove the impedence mismatch. In this strand, we intend to carry out a pilot study to test this hypothesis. Of course, a complete solution to the problem would be beyond the scope of this small project; but if the pilot exercise is successful, we intend to follow it up separately.

6 CONCLUSION

The project has been funded by the UK Engineering and Physical Sciences Research Council for 42 months, starting in November 2006, supporting a postdoctoral researcher (Oliveira), working mainly on generics, and a doctoral student (Wang), concentrating on linguistic mechanisms for indexing. This paper sets out our initial vision for the project; we welcome interest and interaction from the wider community.

REFERENCES

[1] Jim Apple. GADTs are expressive, 7th Jan 2007. Haskell Café mailing list.

[2] Richard S. Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *LNCS 1422: Proceedings of Mathematics of Program Construction*, pages 52–67, Marstrand, Sweden, June 1998. Springer-Verlag.

[3] Corrado Bohm and Alessandro Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:85–114, 1985.

[4] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Haskell Workshop*, pages 90–104, 2002.

[5] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. Submitted for publication, April 2006.

[6] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.

[7] Jeremy Gibbons. Patterns in datatype-generic programming. In *Declarative Programming in the Context of Object-Oriented Languages*, 2003.

[8] Jeremy Gibbons. Design patterns as higher-order datatype-generic programs. In *Workshop on Generic Programming*, 2006.

[9] Jeremy Gibbons and Bruno C. d. S. Oliveira. The essence of the Iterator pattern. In *Mathematically-Structured Functional Programming*, 2006.

[10] Douglas Gregor, Jaakko Järvi, Jeremy G. Siek, Gabriel Dos Reis, Bjarne Stroustrup, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Object-Oriented Programming, Systems, Languages, and Applications*, October 2006.

[11] Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, Cornerstones in Computing, pages 245–262. Palgrave, 2003. ISBN 1-4039-0772-2.

[12] Ralf Hinze. Generics for the masses. In *International Conference on Functional Programming*, pages 236–243, 2004.

[13] Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. In Roland Backhouse and Jeremy Gibbons, editors, *LNCS 2793: Summer School on Generic Programming*, pages 1–56. Springer-Verlag, 2003.

[14] Ralf Hinze and Andres Löh. 'Scrap your Boilerplate' revolutions. In *Mathematics of Program Construction*, 2006.

[15] Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. 'Scrap your Boilerplate' reloaded. In *Functional and Logic Programming*, 2006.

[16] Johan Jeuring and Patrik Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *LNCS 1129: Advanced Functional Programming*, pages 68–114. Springer-Verlag, 1996.

[17] Andrew Kennedy and Claudio V. Russo. Generalized algebraic data types and object-oriented programming. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 21–40, 2005.

[18] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Types in Language Design and Implementation*, 2003.

[19] Andres Löh and Ralf Hinze. Open datatypes and open functions. In *Principles and Practice of Declarative Programming*, March 2006.

[20] Jacob Matthews, Robert Bruce Findler, Paul T. Graunke, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring software for the web. *Journal of Automated Software Engineering*, 2004.

[21] Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer-Verlag, 2005.

[22] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, to appear.

[23] John Meacham. DrIFT homepage. http://repetae.net/ john/computer/haskell/DrIFT/, 2004.

[24] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *ACM SIGMOD International Conference on Management of Data*, page 706, 2006.

[25] T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions, 2002.

[26] George C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, 1997.

[27] Bruno C. d. S. Oliveira and Jeremy Gibbons. TypeCase: A design pattern for type-indexed functions. In *Haskell Workshop*, 2005.

[28] Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. Generics as a library. In *Trends in Functional Programming*, 2006.

[29] M. Parigot. Recursive programming with proofs. *Theoretical Computer Science*, 94(2):335–356, 1992.

[30] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming*, 2006.

[31] Fermín Reig. Generic proofs for combinator-based generic programs. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming*, 2004.

[32] Tim Sheard. Languages of the future. In *Object Oriented Programming, Systems, Languages, and Applications*, 2004.

[33] Tim Sheard. Generic programming in Ωmega. In *Spring School on Datatype-Generic Programming*, 2006.

[34] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *Haskell Workshop*, pages 1–16. ACM Press, October 2002.

[35] Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Types in Language Design and Implementation*, 2007.

[36] Martin Sulzmann, Jeremy Wazny, and Peter J. Stuckey. A framework for extended algebraic data types. In *LNCS 3945: International Symposium on Functional and Logic Programming*. Springer-Verlag, 2006.

[37] The Programatica Team. Programatica tools for certifiable, auditable development of high-assurance systems in Haskell. In *High Confidence Software and Systems*, Baltimore, MD, 2003.

[38] Mads Torgersen. The expression problem revisited. In Martin Odersky, editor, *LNCS 3086: European Conference on Object-Oriented Programming*, pages 123–143. Springer, 2004.

[39] Dimitrios Vytiniotis, Geoffrey Washburn, and Stephanie Weirich. An open and shut typecase. In *Types in Language Design and Implementation*, 2005.

[40] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[41] Philip L. Wadler. The expression problem. Java Genericity mailing list, 12th Nov 1998.

[42] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.

[43] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Principles of Programming Languages*, pages 214–227, 1999.

[44] Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187:147–165, 1997.

A CATALOGUE OF INDEXED PROGRAMMING EXAMPLES

The following is a collection of the most representative examples of indexed programming that we have found in the literature. The examples are classified on the first level by the type of indices; on the second level by the datatypes and functions being indexed; and on the third level by examples of usage.

Enumerations:

- Red-black trees
- State transitions
- Units and dimensions
- SQL injection
- Object ownership

Natural numbers:

Vectors (by size)

- Constant sizes (*reverse*, *safeHead*)
- Arithmetic on sizes (*append*)
- Bounded sizes (list with a fixed maximum size)

Vectors (by element values)

- Inequality on values (insertion into sorted list)

Trees (by height)

- Constant arithmetic (*insert*)
- Variable arithmetic (*merge*)

Matrices (by dimension)

Type representations:

Generic Functions (as directives)

- Print descriptors (indexed by a family of *printf* types)
- Equality (indexed by a family of polynomial types)

Untyped Terms (as type information)

- Well-typing proof

Other datatypes:

Beta-Reduction Rules (as input and output terms)

- Termination Proof

Regular Expression Matching (as regular patterns)

- Matching proof