A comparison of C++ concepts and Haskell type classes

Jean-Philippe Bernardy Patrik Jansson Marcin Zalewski Sibylle Schupp Andreas Priesnitz

Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg
{bernardy,patrikj,zalewski,schupp,priesnit}@chalmers.se

Abstract

Earlier studies have introduced a list of high-level evaluation criteria to assess how well a language supports generic programming. Since each language that meets all criteria is considered generic, those criteria are not fine-grained enough to differentiate between languages for generic programming. We refine these criteria into a taxonomy that captures differences between type classes in Haskell and concepts in C++, and discuss which differences are incidental and which ones are due to other language features. The taxonomy allows for an improved understanding of language support for generic programming, and the comparison is useful for the ongoing discussions among language designers and users of both languages.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

Keywords Generic Programming, Specification-Driven Development, Concepts, Type Classes, C++, Haskell

1. Introduction

Generic programming is a programming style that cross-cuts traditional programming paradigms: its features can be traced in languages from different provenances, and there exist many definitions for it, as Gibbons [8] noted. In the Haskell community, *datatype-genericity* is normally the most important one. However, in this paper we are interested in *property-based* generic programming: both Haskell type classes and C++ concepts support it. It is remarkable that these languages, normally considered far apart, are closely related when it comes to their support for generic programming.

Folklore has it that C++ concepts are much like Haskell classes, that Haskell classes are very similar to concepts, or that the two correspond to each other—which all is true but rather vague. The goal of this paper is to work out in detail how Haskell type classes and C++ concepts are similar, and in what way they differ.

Similar comparative work has been done earlier. An "extended comparative study of language support for generic programming" [7] compared not just Haskell and C++, but a total of 8 languages, based on a set of recommended language features for generic programming that served as evaluation criteria.

However, both Haskell and C++ are changing. The current development version of GHC (the Glasgow Haskell Compiler) provides new features for associated types and, perhaps more dramatically, the next version of C++ will turn concepts from mere documentation artifacts to a full-fledged language feature [6, 14]. Hence, when we refer to Haskell we mean the language implemented in the upcoming 6.10 release of GHC, and C++ refers to the concepts proposal [15] currently considered by the C++ standardization committee.

Due to those developments, the deficiencies that the earlier comparison revealed—in particular on part of C++—no longer exist: C++ now provides as much support for concepts, associated types, and retroactive modelling ([7], Table 1) as Haskell does. If one were to apply the previous evaluation criteria again, Haskell and C++ would, thus, be indistinguishable as generic-programming languages. The differences that undoubtedly exist between the two languages therefore call for a refinement of the previous evaluation criteria, which we provide in this paper.

We take the previous taxonomy [7] as starting point and identify more fine-grained differences in the syntax and semantics of concepts, modellings, and constraints. While the previous comparison found that C++ lacked support for 5 of the altogether 8 evaluation criteria and discussed various workarounds, we reckon that today all but one of the criteria are met. Instead, we focus on the different ways in which they are supported in the two languages. Our guiding question thereby is not just where, but also why differences exist. It is particularly important to understand whether the design decisions motivating the differences are intrinsic to each of the languages, and where it is possible for one language to adopt a feature that the other language introduced. As we show, many design details are rooted in other major language features: in C++, many decisions stem from the motivation to integrate concepts with the existing mechanisms for overloading, while many Haskell decisions are motivated by support for type inference. Yet, we also found that each language could incorporate some features of the other language, and thus each could improve both the expressivity of its generic-programming facilities and the convenience with which they can be used.

In summary, our contributions, condensed in table 2, are:

- a refined set of criteria for evaluation of property-based generic-programming language facilities;
- refined answers to the questions posed by Garcia et al., updated to the latest versions of C++ and Haskell;
- a distinction between accidental and necessary differences, and some suggestions on how to cross-breed Haskell and C++.

2. Background and Terminology

2.1 Terminology

The same generic-programming feature is often named differently in different languages; our two subject languages, Haskell and C++, each come with their own vocabulary. To reduce confusion, we follow the terminology introduced by Stepanov and Austern for

WGP'08 080920, Victoria, BC, Canada.

```cpp
// concept
concept Hashset<typename X> : HasEmpty<X> {
  typename element;
  requires Hashable<element>;
  int size(X);
}
// modelling
template<Hashable K>
concept_map Hashset<intmap<list<K>>> {
  typedef K element;
  int size(intmap<list<K>> m) { ... }
}
// algorithm
template<Hashset T>
bool almostFull(T h) { ... }

// instantiation
intmap<list<int>> h;
bool test = almostFull(h);
```

(a) C++

```haskell
-- concept
class (HasEmpty x, Hashable (Element x))
    => Hashset x where
  type Element x
  size :: x -> Int

-- modelling
instance Hashable k
      => Hashset (IntMap (List k)) where
  type Element (IntMap (List k)) = k
  size m = ...

-- algorithm
almostFull :: Hashset t => t -> Bool
almostFull h = ...

-- instantiation
h :: IntMap (List Int)
test = almostFull h
```

(b) Haskell

Figure 1. Example of the same generic code in C++ and Haskell

```cpp
auto concept EqualityComparable<typename T, typename U = T>
{
  bool operator==(T t, U u);
  bool operator!=(T t, U u) { return !(t == u); }
}
```

(a) C++

```haskell
class  Eq a  where
  (==) :: a -> a -> Bool
  x == y    = not (x /= y)

  (/=) :: a -> a -> Bool
  x /= y    = not (x == y)
```

(b) Haskell

Figure 2. Example of similar concepts from the standard libraries of C++ and Haskell

C++ [1, 36], which Garcia et al. mapped to Haskell [7]. Table 1 summarises it, updated with the new terminology from C++. The rest of this section provides brief definitions and a short illustration of the core terms in both languages. We expand on the definitions in the later sections.

A *concept* can be considered an abstract specification of a set of types. Semantically, a concept has two aspects:

1. It corresponds to a *predicate* over types. When a type satisfies this predicate, we say that it is a *model* for the concept. Intuitively, such a type meets the concept specification.
2. It has a number of *associated entities*: values, functions, or types[1]. Definitions for these entities are provided separately, for all models of the concept.

A concept C_1 is said to *refine* concept C_2 when the predicate of C_1 implies the predicate of C_2. In other words, the models of C_1 form a subset of the models of C_2. As a corollary, the associated entities of C_2 are available whenever entities of C_1 are available.

Modelling declarations define which types satisfy the concept predicate and assign implementations to the associated entities when indexed at a type that is covered by the modelling.

Generic algorithms are traditionally parametric over the types they accept, and thus correspond to template functions in C++ and polymorphic ones in Haskell. We are particularly interested in concepts, which can be used to *constrain* the type parameters to those algorithms, and conversely make the associated entities available. A generic algorithm can then be *instantiated*, that is, applied to concrete types. Using type system terminology, the generic algorithm is polymorphic, whereas its *instantiation* is monomorphic: it can only be used for the types given.

[1] A fourth kind of associated entity are axioms (see [15]), but we choose to leave them out of the discussion.

Term	Haskell	C++
concept	type class	concept
	`class`	**`concept`**
refinement	subclass	refinement
	`=>`	**`:`**
modelling	instance	concept map
	`instance`	**`concept_map`**
constraint	context	requires clause
	`=>`	**`requires`**
algorithm	polymorphic function	function template
	(no keyword)	**`template`**

Table 1. Terminology and keywords

2.2 Examples

The two examples in figures 1 and 2 give a general idea of how generic code is written in C++ and Haskell. In the following, text typeset in *italics* refers to concepts in a language-independent context, while text typeset in `teletype` refers to concrete syntax of C++ or Haskell.

In figure 1, we show a hypothetical *Hashset* concept, a modelling, a generic algorithm constrained by that concept, and an admissible instantiation of the algorithm. The code in the first example has corresponding semantics in C++ and Haskell, so that one

	§	Criterion	C++	Haskell	Definition
Concept	3.1	*Multi-type concepts*	yes	yes	Multiple types can be simultaneously constrained
	3.1	*Multiple constraints*	yes	yes	More than one constraint can be placed on a type parameter
	3.2.1	Type-functions as parameters	yes	yes	Concepts accept type-level functions as parameters
	3.2.2	Value-level parameters	some	⇆	Concepts accept value-level parameters
	3.2.3	Defaults for parameters	yes	⇆	Default values can be given to concept parameters
	3.2.4	Functional dependencies	⇆	yes	Type parameters can be specified to be functionally dependent
	3.3	*Type aliases*	yes	yes	A mechanism for creating shorter names for types is provided
	3.3	Concepts aliases	some	⇆	A mechanism for creating shorter names for concept expressions is provided
Modelling	4	*Retroactive modelling*	yes	yes	New modelling relationships can be added after a concept has been defined
	4.1	Monomorphic modelling	yes	yes	Monomorphic types can be models
	4.1.1	Parametric modelling	yes	yes	Modellings can be universally quantified
	4.1.2	Overlapping modelling	yes	yes	A type can model a concept via two modellings or more
	4.1.3	Free-form modelling	yes	yes	Free-form contexts and heads can be used in modellings
	4.2	Default definitions	yes	yes	Associated entities can be given default definitions in concept definition
	4.2.1	Structure-derived modelling	⇆	some	Modellings can be generated by structure of type definition
	4.2.2	Modelling lifting	⇆	yes	Modellings can be generated by lifting through wrapper types
	4.2.3	Modelling from refinements	yes	⇆	Modelling of refining concept define modellings for refined concepts
	4.2.4	Implicit definitions	yes	⇆	Default modellings, definition of associated entities by name/signatures
	4.2.5	Automatic modelling	yes	⇆	For a concept, modellings are generated automatically
Constraint	5.1	Constraints inference	some	some	Constraints to generic algorithm can be deduced
	5.2	*Associated types access*	yes	yes	Entities associated to concepts can be types
	5.3	*Constraints on associated types*	yes	yes	Concepts may include constraints on associated types
	5.3	Type-equality constraints	yes	yes	Type variables can be constrained to unify to the same type
Algorithm	6.1	*Implicit argument deduction*	yes	yes	The arguments for the type parameters of an algorithm can be deduced
	6.2	Concept-based specialisation	yes	no	Algorithms can be specialised on existence of modelling
	6.3	Separate type-checking	yes	yes	Generic functions can be type-checked independent of calls to them
	6.3	*Separate compilation*	⇆	yes	Generic functions can be compiled independent of calls to them

Table 2. Comparison criteria and their support in C++ and Haskell. The first column gives the section where the feature is discussed. Italics is used for features previously [7] introduced. Double arrows (⇆) denote a missing feature that can be ported from the other language.

can transpose knowledge of one syntax to the other. Given the correspondence provided in table 1, the following discussion applies to both Haskell and C++ code. The *Hashset* concept has one parameter x, and *Hashset* refines *HasEmpty*. The body of the concept introduces an associated type, *element*, and an associated function, *size*. The associated type is required to be hash-able. The subsequent modelling is quantified over the type parameter k and states that $intmap(list(k))$ models the *Hashset* concept for every k that models *Hashable* (see section 4.1.1). The body of the modelling states the values for the associated types and functions of the *Hashset* concept: the type *element* is implemented by aliasing the type parameter k, and the function *size* is given a definition, in the ellipsis. Then we show a generic algorithm *almostFull*, whose type parameter T is constrained by the *Hashset* concept. In Haskell the constraint is written before `=>` in the type signature, while in C++ it is expressed by replacing the usual `typename` keyword preceding a type parameter with the name of the constraining concept. In the remainder of the paper we use this "predicate" syntax, but in some examples we either augment it or replace it with a more explicit constraints syntax: a requirements clause signified by the keyword `requires`. Finally, in the last two lines, we show an instantiation of *almostFull* where the type argument, $intmap(list(int))$, is left implicit.

In figure 2, we show two variants of the concept of equality, which we use throughout the paper to discuss various language features. Figure 2(a) shows the C++ `EqualityComparable` concept as given in the ConceptGCC [11] compiler. We often abbreviate it to *EqComp* in the text. The concept *EqComp* has two parameters, of which the second is by default set to the first, and two associated functions, of which the second is by default defined in terms of the first. The keyword `auto` indicates that it supports automatic modelling (explained in section 4.2.5). The two type parameters allow for comparison of values of different types. This allows comparing apples and oranges, but a more typical examples would be comparing apples and references to apples with a modelling like $EqComp(T, \&T)$. Figure 2(b) shows the `Eq` concept from the Haskell prelude [28]. The `Eq` concept has one parameter and two associated functions, each with a default definition given in terms of the other. The defaults mean that it is enough to define the more convenient of the two methods in each modelling; if neither is defined, they will both be non-terminating.

2.3 Background

Both Haskell type classes and C++ concepts arose from the need to structure ad-hoc polymorphism [3], and therefore turned out very similar to each other. Yet, type classes were introduced in the early stages of Haskell design to support overloading, while concepts were introduced rather late to C++, providing separate type-checking for the existing generic-programming techniques without incurring extra run-time costs.

The different starting points have influenced how concepts and type classes are implemented. C++ requires instantiation of (function and class) templates at compile time, for each collection of arguments; introduction of concepts preserves the compile-time instantiation semantics. In contrast, Haskell type classes are traditionally implemented by dictionary passing in a class-less language, as pioneered by Wadler and Blott [41]. Strictly speaking, however, the Haskell report [28] does not prescribe any particular implementation.

2.4 Evaluation criteria

Beyond the fundamental difference in motivation and approach, which is detailed in section 3.1, we identify many points of comparison between Haskell and C++ concept abstractions, and broke

them down along the terms introduced in section 2.1. Table 2 summarises the criteria.

In section 3, we examine how concepts in general and their parameters in particular are treated in each language. In sections 4 through 6, we focus sequentially on modellings, constraints, associated types, and generic algorithms.

3. Concepts

3.1 Concept-checking

From a high-level perspective, type-checking of concepts in C++ and type classes in Haskell is very similar. On the one hand, an algorithm may use the associated entities of a concept, and thus require certain modellings to be defined for the types for which it is instantiated. In general, there is no limitation on the constraints that can be put on an algorithm. The compiler, on the other hand, ensures that algorithms are only instantiated to those types for which the required modelling exists. The entities of the modellings chosen to fulfil the constraints are then used by the instantiated algorithm. Finally, we note that both languages support concepts with multiple type arguments, in which case the concept describes a relation between them.

However, the differences in approach outlined in section 2.3 have consequences on the particulars of the compiler checks. We now outline how checking of concepts and their usage work in both languages.

Haskell In Haskell, the constraints of an algorithm are inferred from its definition. Each usage of an associated entity or of another constrained algorithm induces the corresponding constraints. Using known modellings and refinements, these constraints are then simplified, and checked against those provided by the programmer if a type signature is provided. If a constraint is discharged during the simplification phase, it has to be done using a particular modelling. This modelling determines which version of the associated entities will be used at run time.

When a generic algorithm is instantiated, the compiler tries to infer its type arguments. If the compiler does not succeed, it deems the call ambiguous and rejects the code.

C++ In C++, the programmer specifies constraints explicitly when defining a generic algorithm. These constraints make the associated entities available in the definition of the algorithm. Constraints are not inferred from the definition of the algorithm, but some constraints can be propagated from the types used in the algorithm signature (see section 5.1). Note that the compiler does not refer to the set of modellings known at this point: modellings are not used to extend the set of constraints or to provide more symbols implicitly. Still, concept refinements are taken into account: it suffices for the programmer to specify the most refined concept.

If a generic algorithm is instantiated to a monomorphic type, the compiler decides which modellings to use, and checks the instance again. If a type-parameter is polymorphic, the function continues to be a template, and the above applies. When a generic algorithm is instantiated, its type parameters are inferred, if possible. As a fall-back, the programmer can explicitly state them using angle brackets: `almostFull<T>(x)`.

We can summarise the differences as follows:

- Haskell deduces the constraints from the associated entities used, C++ does the converse: it deduces the entities available given the constraints provided by the programmer.
- Haskell infers constraints when the type of the function is not provided, C++ only propagates constraints arising from the signature.
- Haskell uses the "current set of known modellings" at every definition to simplify constraints. C++ resolves modellings at instantiation time.

3.2 Concept parameters

In their simplest form, concepts have one type parameter. More evolved forms include multi-parameter concepts and type-function or value-level parameters, and allow default bindings. We will discuss these extensions along with functional dependencies.

3.2.1 Type-functions as parameters

Since version 1.3, Haskell offers "constructor classes" [19]. This means that concepts can not only apply to types, but also to *type constructors*. This feature has proven very useful in practise to model universally polymorphic concepts, like `Functor` or `Monad`.

```
1 class Functor f where -- f :: * -> *
2   fmap :: (a -> b) -> f a -> f b
3
4 instance Functor (List a) where
5   fmap f Nil         = Nil
6   fmap f (Cons a as) = Cons (f a) (fmap f as)
```

C++ does not offer type-constructors as such, but provides type-functions in the form of templates. Therefore, type-constructor parameters in concepts directly correspond to "template-template" parameters, and we can translate `Functor` to[2]:

```
1 concept Functor<template<typename> class F> {
2   template<typename A, typename B>
3   function1<F<A>, F<B>> fmap (function1<A,B>);
4 }
5
6 template<typename A, typename B>
7 class list_func {
8 public:
9   function1<A, B> f;
10  list_func(function1<A, B> f_in) : f(f_in) { }
11
12  list<B> operator()(list<A> la) {
13    list<B> temp;
14    // apply f to each element in la
15    // and put the results in temp
16    transform(la.begin(), la.end(),
17              back_inserter(temp), f);
18    return temp;
19  }
20 };
21
22 concept_map Functor<list> {
23   template<typename A, typename B>
24   function1<list<A>, list<B>> fmap(function1<A, B> f)
25   {
26     return list_func<A, B>(f);
27   }
28 }
```

However, template-template parameters are not frequently used, because the parameter list of a template that is used to substitute a template-template parameter must match the parameter list of that template-template parameter exactly [40]. Since most C++ templates have many parameters, matching the concept signature is very difficult. Further, this problem cannot be fixed by partially applying templates, because C++ offers no syntax to do this. This

[2] In this example we use the `function1` template from the Boost libraries[30], which provides a function type.

problem occurred for template definitions in the past, and was solved by using templates nested in classes ("member templates"). A corresponding encoding can be used for concepts, since they also allow member templates.

We should note that Haskell restricts the type functions that can be defined: the language of type-level expressions is purely applicative. This has an impact on a generic programming point of view: it means that some type-functions cannot be made models of any concept.

```
data Map key value = ...
instance Functor (Map k) where ...
```

If we instead had the opposite order of the arguments: `data Map value key` then the `Functor` modelling would become:

```
instance Functor (Λvalue -> Map value k) where ...
```

but Haskell has no type level Λ, and indeed unification is only first-order, so this is invalid.

3.2.2 Value-level parameters

In C++, concepts can also be parametrised over values instead of types:

```
concept Stack<typename T, int size> { ... }
concept_map Stack<char, 512> { ... }
```

In both Haskell and C++, types and values are completely separate universes, therefore no run-time value could influence the selection of a value-dependent modelling. This C++ feature can therefore be emulated in Haskell by duplicating the structures of value-level at the type-level [24]. We argue in favour of porting the C++ feature to Haskell: the syntax of type-level expressions is not suited to this kind of trick, and thus one risks producing unreadable programs by using them. Providing a full-fledged two-level language as Sheard [33] did would be a major change, but limiting it to some scalar types as in C++ is simpler and still useful.

3.2.3 Defaults for parameters

In C++, the last few parameters of a concept (it can be the entire parameter list) can be given defaults. When referring to such a concept, some or all of those parameters may be omitted. As an example (based on figure 2), uses of $EqComp(T)$ are treated as $EqComp(T,T)$. Widespread usage of multi-parameter concepts is envisioned for the future C++ standard library, and defaults for parameters are important to reduce the tedium of using such concepts.

Haskell does not support defaults for concept parameters, but the feature could be easily ported. However, we should warn against careless usage of default parameters: as we discuss in section 4.1.3, limiting the number of type-variables in constraints is essential to ensure termination of type-checking. Default parameters can hide type-variables in such contexts, and therefore make type-checking termination problems more difficult to track down.

3.2.4 Functional dependencies

Jones [20] describes an extension to the Haskell type system where users can specify functional dependencies between the arguments of a given concept. In the example below, we give an alternate definition of the *Hashset* concept of figure 1. Instead of having *element* as an associated type, it becomes an extra parameter, with a functional dependency stating that *element* is uniquely determined by *set*.

```
class Hashset set element | set -> element where
  ...
```

This means that when $Hashset(set, element)$ holds, a given type for *set* yields a unique, concrete type for *element*: a modelling declaration that violates this property will be rejected. If the functional dependency was not present in the multi-parameter concept $Hashset(set, element)$, the dependency would be implicit by the non-existence of some models.

Making the restriction explicit in the the concept declaration allows the compiler to use that knowledge, improving the type checking of generic algorithms: inferred types become more precise and type errors can be detected early. In some cases, the compiler can even accept a declaration that it would have to reject without the functional dependency.

Despite thorough analysis [39], functional dependencies remain a controversial feature: Chakravarty et al. [4] argues that associated types provide much of the functionality but cause less complexity in the type system.

Indeed, C++ chose to ignore this feature altogether. Further, since constraints are not inferred, the usefulness of the feature would be even more limited.

3.3 Concept aliases

Garcia et al. [7] recognise that type aliases are an important feature for generic programming: abbreviating long type expressions reduces clutter, and the ability to define abstractions improves maintainability. Both Haskell and C++ acknowledge this and provide type aliases.

One would then naturally expect aliases for concept-level expressions. Refinement provides this feature to some extent. For example it is possible to define a concept *BoundedLattice* refining *Lattice* and *Bounded*; $BoundedLattice(a)$ can then be used in constraints in place of $Lattice(a), Bounded(a)$. In C++, the mechanisms for modelling from refinement, parametric modelling, and automatic modelling (covered in section 4) give a rich set of possible encodings for concept aliasing (see [9] for details). For example, giving a modelling of *BoundedLattice* can also define modellings for *Lattice* and *Bounded*, as we explain in section 4.2.3. Concept aliasing has been proposed for Haskell [25, 37] as an explicit language construct, rather than by reusing or extending existing features.

In addition to concept aliasing, constraint aliasing has been proposed for C++ [2]. Constraints in C++ have to be written out explicitly, and parts of them are often repeated when associated types are constrained. To reduce repetition, a part of a constraint can be named for later reference.

4. Modelling

Given native support for concepts, the relation between specification and implementation is non-intrusive: it is possible to add modelling relationships after a type or a concept has been defined; and it is possible to add concepts and their modellings after types have been defined. Beyond basic support for retroactive modelling, advanced modelling features found in C++ and Haskell can be grouped in two categories that we named modelling flexibility and modelling shortcuts. The first category comprises parametric modelling, overlapping and free-form modellings as degrees of flexibility allowed in the way a modelling can specify the set of types it applies to. The second category comprises language constructs with which users conveniently define new modellings, and guide compiler-generated modellings. We elaborate on each feature.

4.1 Modelling flexibility

The simplest way to define modellings, trivially supported by both Haskell and C++, is to state that a monomorphic type models a concept, and to supply the value of the methods and other associated members. For example, if the proper concept has been previously

defined, stating that Booleans can be compared for equality is done with a simple modelling declaration: `instance Eq Bool where ...` in Haskell or `concept_map Eq<bool> {...}` in C++. In this section, we prefer a language-independent notation where we also name modellings: $M_1 \equiv Eq(bool)$.

Such monomorphic modellings are very easy to deal with: constraints can be simplified only if all the concept parameters are monomorphic, and therefore the concept-checking behaviours of Haskell and C++ are almost identical in such a limited context.

4.1.1 Parametric modelling

Monomorphic modelling is quite restrictive, however, and one often wishes to define modellings parametrically. For example, we may want to state that all lists of a certain type are comparable: $M_2 \equiv \forall a.Eq(list(a))$; or more realistically, lift comparison on elements to comparison on lists, using a constraint: $M_2 \equiv \forall a.Eq(a) \rightarrow Eq(list(a))$. The left hand side of $\rightarrow$ in this notation is called *context*, or assertion, whereas the right hand side is the *head*. Note that there can be only one arrow: both in C++ and Haskell the assertion must be a simple conjunction of concepts.

Parametric modellings are supported both in Haskell and C++. In C++, quantification is denoted by the `template` keyword, while in Haskell all free variables are implicitly universally quantified at the top-level.

4.1.2 Overlapping modelling

One might want to have a parametric modelling for most cases and specialise it for some cases, for performance, better customisation, or other reasons. A typical example in Haskell, is that one wants a display function for all lists, and a specialised version for lists of characters, which displays them as strings: $M_l \equiv \forall a.Show(a) \rightarrow Show(list(a))$, $M_s \equiv Show(list(char))$. For $list(char)$, both modellings M_l and M_s apply: we say that they *overlap*. In such a case, the language must define which modelling is preferred. Both Haskell and C++ try to use the most specific modelling.

Most Haskell implementations allow overlapping modellings. When a constraint is discharged, the dictionary of the most specific modelling is used. In C++, the most specific modelling is chosen upon template instantiation, and this usage pattern is called concept map specialisation.

Sometimes, in presence of multi-type concepts, a most specific modelling does not exist. For example, given $M_1 \equiv \forall a.C(int,a)$ and $M_2 \equiv \forall a.C(a,int)$, the constraint $C(int,int)$ can obviously be satisfied using either M_1 or M_2, but neither is more specific than the other. An instantiation using the concept at (int,int) will therefore be rejected: the modelling to use is ambiguous. This behaviour is the same in C++ and in Haskell with overlapping instances.

In Haskell, overlapping instances raise issues not only at instantiation, but every time constraints are simplified. Indeed, every time a constraint is simplified out, a specific modelling has to be chosen. The situation is more complex than in the monomorphic instantiation case explained above, because type variables can be unified to concrete types only partially: the difficulty is to make sure that, when the type variables become fully unified to concrete types in further instantiations, the most specific modelling will then be used. This tricky issue is discussed in more detail by Peyton Jones et al. [29]. Since C++ does not try to simplify constraints, this difficulty does not arise.

4.1.3 Free-form modelling

If no restriction is placed on the form of contexts and heads of modellings, the modelling language is very powerful: it is possible to express very complex properties of types. Conversely, one then faces the problem of undecidability of the concept-checking algorithm. In Haskell indeed, solving a set of constraints given a set of rules (given by modellings) can be potentially non-terminating: using a parametric modelling that has non-trivial assertions can make the set of constraints bigger, and if the rule can be applied again, the solver will never converge. In C++, non-termination can also occur in a corresponding situation. When trying to instantiate a template function, the compiler will use the modellings to lookup possible assignments to the type parameters. If those yield infinitely many possibilities for the type assignments, the lookup will not terminate if none is valid.

To prevent this unfortunate occurrence, one can restrict the form modellings can take. In Haskell 98, valid modellings are of the form $\forall a_1 \ldots a_n.C_r a_i \rightarrow C_m(t(a_1,\ldots,a_n))$ where t is a type constructor, and $a_1 \ldots a_n$ are *distinct* type variables. While this is sufficient to ensure that a terminating algorithm exists, a number of more flexible strategies can be adopted. For example, GHC offers the following rule: 1. No type variable has more occurrences in the assertion than in the head; 2. The assertion has fewer constructors and variables (taken together and counting repetitions) than the head.

The C++ community is less concerned by the undecidability issue; C++ type-checking is already undecidable because of templates. While this may sound very dangerous, programmers are already used to non-termination in the realm of values, and have proven to be able to apply their intuition in the realm of types. Indeed, Haskell programmers have also found that disabling termination checking can be very useful, in order to encode complex type rules as constraints and modellings [24].

Finally, we note that C++ modellings are not completely free-form: at least one argument of the modelled concept must be specialised, either by being unified with a type-constructor or restricted by a concept.

4.1.4 Summary

Flexibility in modelling has a price: overlapping modellings can yield ambiguity, complex assertions can bring undecidability. The Haskell community is conservative in this respect and disable this by default. While C++ gives almost full flexibility, knowledge of the trade-offs can be useful to programmers, so that complexity and the ensuing costs are understood.

4.2 Modelling shortcuts

Specifying modelling using the above constructs can sometimes be quite repetitive and therefore both C++ and Haskell both provide syntactic sugar to define them concisely. Interestingly, beyond the ability to give default definitions to associated entities in the concept declaration, those mechanisms turn out to be entirely different in either language.

4.2.1 Polytypic modelling

While concepts aim at capturing ad-hoc genericity, modellings themselves are often datatype generic [8]. For some concepts, deriving the modelling from the structure of types is sensible most of the time, but for some types the programmer wants to give their own modelling. For example, structural equality can be used most of the time, but does not make sense for some data structures, like balanced trees.

Haskell offers a mechanism for deriving modellings from type structure:

```
1 data Bool = False | True
2   deriving Eq
```

Until recently, derived modellings have been coupled with type definitions but that restriction has been lifted since GHC 6.8, enabling retroactive derived instances (also called *stand-alone deriving*).

Unfortunately, the standard restricts the `deriving` construct to a few predefined concepts (*Eq*, *Show*, etc.). A few generalisations

(involving some extension to the Haskell language) have been implemented [16, 17]. Notably, Template-Haskell [34] provides a generic mechanism that suffices to implement a customised derivation construct [26]. There are also around ten proposals for the design of a generic library of concepts supporting polytypic modelling in Haskell (see Rodriguez et al. [31] for a comparison).

C++ supports certain operations in a similar way; for example, a compiler automatically defines the equality operator == for every type, unless the operator is explicitly given. Together with automatic modelling, discussed later in this section, the generated operator == also automatically produces modellings for the *EqComp* concept (from figure 2). However, in general, the structure of a type in C++ is much less informative than in a functional language such as Haskell; a type often reflects low-level details of implementation, while, in a functional language, the structure of an algebraic data-type more often reflects the intended, logical functionality of the type. Munkby et al. [27] discuss the issue in more detail and they propose *interface traversal* as a way to improve usability of automatic constructs that depend on the structure of a type in an imperative language such as C++.

4.2.2 Modelling lifting through wrapper types

Both Haskell and C++ provide *type aliases* to name and reuse type expressions. An example is `type IM = IntMap (List Int )` corresponding to `typedef intmap<list<int>> IM`. In addition to that, the Haskell `newtype` construct allows defining a type that has exactly the same representation as another, but is treated as a completely separate type by the type checker. C++ does not offer a construct equivalent to `newtype`, but one can still define a wrapper structure by hand, without any special support from the language.

Since wrappers define fresh types, they do not inherit the modellings of their wrapped type. This can be problematic, in the case where the wrapper is there to change only few aspects of the wrapped type: one would like to reuse most modellings, but they must be given an explicit definition. To facilitate this process, Haskell provides the `newtype deriving` construct to lift modellings of given concepts to a `newtype`. A mere mention of the concept to model is enough to specify the whole modelling.

```
1 newtype Age = Age Int
2   deriving (Hashable)
```

Modelling lifting works only when it has the form `newtype TC` $v_1 \dots v_n$ `= T (`$t\ v_{k+1} \dots v_n$`) deriving (`$C_1 \dots C_m$`)`, where C_i are partially applied concepts, with the restriction that the type variables v_i must not occur in t nor C_i. This restriction ensures that the implicit definition of the modelling makes sense.

There is no C++ construct corresponding to `newtype deriving` that transposes modellings of the wrapped type to the wrapper, but the only difficulty to port it is that `newtype` has no equivalent.

4.2.3 Modelling from refinement

In C++, refined concepts can be separated into two categories, each with different modelling semantics. The `Hashset` concept from figure 1 demonstrates the difference:

```
1 concept Hashset<typename X> : HasEmpty<X> {
2   typename element;
3   requires Hashable<element>;
4   int size(X);
5 }
```

When a modelling for `Hashset` is provided, the modelling for `Hashable` must be provided beforehand while the modelling for `HasEmpty` is generated by the compiler if it has not been given before. For that reason, the programmer may provide definitions for the associated entities of the concept `HasEmpty` in the modelling declaration for `Hashset`; these definitions are then used to generate a modelling for `HasEmpty` when necessary. Since modellings for `Hashset` may be already defined, the semantics of C++ concepts gives *compatibility* rules by which definitions in the refining and refined concepts are checked for conflicts. The support for modellings from refinements does not affect the power of concepts but it greatly simplifies use of concepts in practise—in large libraries refinements are often more numerous than in our simple example. While there is no corresponding mechanism in Haskell, it should be possible to add a similar feature and it could have the same effect of easing the use of type classes in practise as it has for concepts in C++.

4.2.4 Implicit definitions

In C++, a concept map may leave out the definitions of associated entities, making the definitions *implicit*. Implicit definitions are filled in by the compiler. In a nutshell, default definitions are considered first and, if no default definition is available, entities required by the modelled concept are looked up in the enclosing scope. Entities whose name and types match those given in the concept declaration are bound to the associated entities automatically. To allow optimisations similar to these possible with unconstrained C++ templates, the generation process is actually more complex (see Gregor [12]) but we elide the discussion here. We give a simple example based on the concept from figure 2 where `!=` has a default implementation in terms of `==`:

```
1 concept_map EqualityComparable<int> {/* implicit */}
```

The modelling $EqComp(int)$ leaves all definitions implicit. The definition of == is deduced by the compiler, since `int` has an operator == with matching type. Then `!=` can use its default implementation, forwarding to ==.

Because in Haskell a scope cannot contain functions with the same name as an associated function in one of the type classes in the scope, direct porting of definitions generation as performed in C++ is not possible. The mechanism could be adapted, though, if definitions could be taken from a scope *different* than that of a type class for which modelling is defined. For example, a pre-existing set structure, defined in a module `OldSet`, could be retrofitted into our `Hashset` concept:

```
1 instance Hashset OldSet.Set where
2   import OldSet -- possible future syntax
```

4.2.5 Automatic modelling

To further alleviate the burden of defining modellings and ease retrofitting of existing types, C++ allows to mark concepts with the keyword `auto`. The compiler then tries to generate modellings automatically for that concept. Note that, in contrast to most other shortcut facilities, automatic modelling works on the level of the concept, instead of type by type.

Modellings for `auto` concepts can still be provided explicitly but, if they are not, the compiler can generate them on demand, when they are actually necessary to instantiate a generic algorithm. In such a case, the compiler uses the default definitions and the implicit definition mechanism to assign values to each associated entity in the concept. If this fails, then the concept predicate is deemed not to hold for that instantiation, and the instantiation is thus rejected.

The following listing shows an example using automatic modelling (again using the concept from figure 2):

```
1 template<EqualityComparable T>
2 bool f(T t1, T t2) { ... }
3
4 bool test = f(1, 2);
```

The listing shows an algorithm `f` where the type parameter `T` is constrained to be *EqComp*. When the algorithm is instantiated to the built-in type `int` on line 4 the compiler will try to generate a modelling *EqComp*(*int*) as if the following declaration had been in scope:

```
concept_map EqualityComparable<int> { }
```

Since `int` has the operator == defined, the implicit definitions feature kicks in and the appropriate definitions of the associated functions are automatically generated as above (section 4.2.4).

A mechanism similar to `auto` would be sensible in Haskell, but with slightly different semantics, since the implicit definition mechanism cannot work with the current scope. Instead, the Haskell version could use the `deriving` mechanism.

4.3 Summary

Property-based generic programming gives great flexibility: types can be be made models of concepts arbitrarily. A cost for this flexibility is that modelling definitions can be lengthy. Hence, features for shortening the definitions of modellings, or omitting them entirely are important. In this area, Haskell and C++ provide different kinds of mechanisms, and each provides a good source of inspiration for improvement of the other.

5. Constraints and associated types

5.1 Constraint inference

We have briefly explained the difference regarding constraint inference in section 3.1: Haskell infers constraints when the type of the function is not provided, C++ only propagates constraints arising from the signature. Here we will refine this statement and give some intuition for the motivation between the difference.

First, we note that Haskell allows the programmer to omit the constraint specification, but only if the type signature is omitted entirely. Also, C++ supports a limited form of inference, called *requirements implication*, in *declarations* of generic algorithms [15, 14.10.1.2]. Basically, if an algorithm declaration is ill-formed because some constraints are missing, the compiler can automatically fill in these constraints. The constraints can be propagated from other algorithms used in the declaration or implied by language constructs. The following examples illustrate requirements implication:

```
1 template<Hashable T> class hashSet { ... };
2
3 template<EqualityComparable T>
4 void g(hashSet<T> s, T value);
```

The use of `hashSet<T>` on line 4 implicitly adds the requirement `Hashable<T>` to `g`.

```
1 template<Hashset T>
2 Hashset<T>::element  h(T&);
```

To make the return type well formed the compiler adds the requirement `Returnable<Hashset<T>::element>` to `h` and similarly the use of a reference to `T` adds `ReferentType<T>`.

Second, we recall that Haskell has no overloading mechanism beside type classes, and this is what makes constraint inference sensible: an identifier identifies the concept it belongs to unambiguously, so one can recover the concept from the associated entity. This is not the case in C++, because of the underlying overload mechanism. An identifier can refer to entities in many concepts, and the process of selecting which one applies is guided by the constraints provided by the programmer. Adding inference of constraint of top of this behaviour would be awkward. Gottschling [10] proposed constraint inference for C++ based on explicit inference declarations provided by the programmer. While an interesting and potentially useful feature, due to its late introduction to the C++ standardization process it will probably not be included in C++.

In summary, we can say that C++ trades inference of types for an extra, more flexible overloading mechanism.

5.2 Associated types

It was previously noted [7] that one can encode associated types in Haskell with functional dependencies. Such an encoding can be cumbersome, and partly in reaction to that observation, an extension to allow associated data types was proposed [5]. Not long after, the full power of associated types was proposed [4], then implemented [32]. In contrast, C++ concepts include associated types from their inception.

In figure 1, *element* is an associated type of *Hashset*. Note that Haskell allows one to specify on which parameters the associated type depends, whereas the closest C++ equivalent implicitly assumes that it depends on all the arguments on the concept.

5.3 Constraints on associated types

It is important to be able to constrain associated types as well as class parameters. For example, converting a hash set to a string requires that its elements can be converted to a string themselves:

```
1 toString :: (Hashset s, Show (Element s)) =>
2             s -> String
```

```
1 template<Hashset S>
2 requires Show<Hashset<S>::element>
3 string toString(S);
```

In particular, type-equality constraints are useful too, for instance to make sure that two hash sets have the same type of element:

```
1 insertAll :: (Hashset s1, Hashset s2,
2               Element s1 ~ Element s2) =>
3              s1 -> s2 -> s2
```

```
1 template<Hashset S1, Hashset S2>
2 requires std::SameType<Hashset<S1>::element,
3                        Hashset<S2>::element>
4 S2 insertAll(S1, S2);
```

An extension for support of type-equality constraints has recently been implemented in GHC and has been studied as an extension to system F [38]. Schrijvers et al. [32] show that their extension keeps the type-system decidable, which is an important property for Haskell extensions.

In C++, type equality is represented by a built-in, compiler-supported concept, `std::SameType<T,U>`. The concept can be used in constraints of generic algorithms, but its models are fixed: none can be declared.

Note that it suffices to specify `SameType<T,U>` to get type-equality between `F<T>` and `F<U>`, where `F` is any type-level function (a template, or any C++ standard type-constructor like pointer, reference, etc.) [15, 14.10.1]. This means that the semantics of `SameType` are similar to that of Haskell's type-equality constraints.

6. Algorithms

6.1 Type arguments deduction

Both Haskell and C++ try to deduce arguments for the type parameters of an algorithm. However, C++ only uses the type information of the value arguments to the function to infer the template (type) arguments. This is to be contrasted with Haskell that also uses the return type information.

This difference has an influence on how concepts are written in practise: in C++ the return type of a function is often either an

associated type (and modellings provide a value for it) or it is also the type of an argument.

6.2 Concept-based algorithm specialisation

One sometimes wishes not only to overload based on a type, but also depending on whether a concept applies to a given type or not [23].

```
instance Show t => Print t where
  print x = putStrLn (show x)
instance           Print a where
  print x = putStrLn "no Show"
```

The above intends to capture that idea: one wishes to use the first modelling when *Show* is available, and fall back to the second otherwise, taking advantage of "most-specific" rules. Unfortunately, this is invalid Haskell: the heads of the two modellings are identical, and it is the only part that is taken in account to choose a modelling in case of overlap.

Translating the above to C++ can work, but the natural way to tackle the problem is to directly overload on the constraints of an algorithm:

```
concept Show<typename T> { string show(T); }

template<Show T>
void print(T x) { cout << show(x)   << endl; }

template<typename T>
void print(T x) { cout << "no Show" << endl; }
```

Porting this feature to Haskell is not possible, because there is no legacy overloading mechanism (beside type classes) to extend.

6.3 Separate type checking and separate compilation

The concept systems of Haskell and C++ both enable type checking of generic algorithms separately from their uses. While in Haskell separate type checking always guarantees safe instantiation of generic algorithms, C++ allows exceptions to separate type checking safety in the interest of generating optimal code. In the current concepts proposal, the safety can be broken by concept-based specialisation, discussed in the previous subsection, or by allowing overload resolution during instantiation (for details see Gregor [12]). These breaches in separate type checking reflect the long-standing practise of choosing the most efficient implementation [18] and are commonly used in the non-concept implementations of STL and other generic libraries.

The Haskell system, furthermore, enables separate compilation: generic algorithms can be compiled to polymorphic object code, that can be applied to its type arguments at run time. This is realised by a dictionary-passing translation [41]. A similar style of compilation could be possible in the context of C++ [13, 14] but it would require changes to the language and would be inconsistent with C++ design goals. On the Haskell side Jones [21] has explored how to compile away the dictionaries to obtain faster code.

7. Related work

There is much literature that analyses concepts, and their implementation in various languages, but to our knowledge only our earlier work [43] compares directly C++ and Haskell.

Peyton Jones et al. [29] explored the trade-offs of various features and choices in designing a concept system. While their results are useful to compare languages from a generic-programming point of view, they focus on an extensions of Haskell type classes, and some of the results need to be reinterpreted to be valuable to a larger community.

Willcock et al. [42] give a language-independent definition of concepts and use that common framework to interpret the implementation of concepts in various languages, including Haskell and C++. As Garcia et al. [7], the version of C++ that they considered did not include linguistic support for concepts.

Siek and Lumsdaine [35] design a type system for concepts as an extension for System F, and identify a number of important features of concept systems in the process. Most of them are mentioned by Garcia et al. [7] or by us, except for *scoped modellings*. Scoped modellings help the programmer to control which modelling to apply. Scope control for modellings is implemented with namespaces in C++, but lacking in Haskell, although named modellings have been studied [22]. We have left a more thorough comparison of this feature for further work.

8. Conclusions and future work

In order to precisely compare C++ and Haskell from a generic programming point of view, we have refined the previous taxonomy of Garcia et al. [7], with respect to the level of support for concepts. More specifically, our criteria capture differences in the way concepts parameters, modelling definitions, constraints, and algorithms are treated in the two languages.

As Garcia et al. identified before, concepts allow for flexible programming of generic algorithms. However, the modelling machinery can be rather verbose, and this creates a tension with one of the driving forces behind generic programming, namely, obtaining concise code. Therefore, it is important to provide modelling shortcuts, and this area of language design has received much attention in both Haskell and C++, as we have seen in section 4.2 Also, inference is very important: omitting inferable details allows the compiler to fill them in "on demand", and this directly allows code to work in more varied contexts. We discussed these aspects in section 5.1 and section 6.1.

Despite the significant differences in the design philosophies of Haskell and C++, we identify a number of features that could be naturally ported from one language to the other: value-level parameters, defaults for parameters, modelling lifting, modelling from refinements, implicit definitions, and automatic modelling.

Other, potentially beneficial, features are possible to port, but in practise they interact strongly with other parts of the language. Therefore, these features are difficult to apply in a context different from the one in which they were introduced; functional dependencies and separate compilation are examples of such features.

Finally, by carefully explaining how concepts are supported in state-of-the-art implementations of generic programming we hope to affect the development of language support for generics in general. The criteria we identified closely map various features of Haskell and C++. While such mapping is important in its own right, for the future it is also important to understand which features are fundamental for generic programming. For example, implicit definitions and refinement seem to overlap with concept aliases. Finding orthogonal comparison criteria will help guiding design of languages with orthogonal generic programming features.

Out of our 27 criteria, summarised in table 2, 16 are equally supported in both languages, and only one or two are not portable. So, we can safely conclude as we started — C++ concepts and Haskell type classes *are* very similar.

Acknowledgments

We thank Gustav Munkby for fruitful discussions on the topic. The anonymous reviewers helped improving the presentation of the paper.

References

[1] M. H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[2] W. Brown, C. Jefferson, A. Meredith, and J. Widman. Named requirements for C++ concepts. Technical Report N2581=08-0091, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Mar. 2008.

[3] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4): 471–523, Dec. 1985.

[4] M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. *SIGPLAN Not.*, 40(9):241–253, Sept. 2005.

[5] M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 1–13. ACM, 2005.

[6] G. Dos Reis and B. Stroustrup. Specifying C++ concepts. In *Proc. 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 295–308, New York, NY, USA, 2006. ACM.

[7] R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek, and J. Willcock. An extended comparative study of language support for generic programming. *J. Funct. Program.*, 17(2):145–205, Mar. 2007.

[8] J. Gibbons. Datatype-generic programming. In *Spring School on Datatype-Generic Programming*, volume 4719 of *LNCS*, pages 1–71. Springer, 2007.

[9] P. Gottschling. Fundamental algebraic concepts in concept-enabled C++. Technical Report 638, Indiana University, 2006.

[10] P. Gottschling. Concept implication and requirement propagation. Technical Report N2646=08-0156, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2008.

[11] D. Gregor. ConceptGCC — a prototype compiler for C++ concepts. `http://www.generic-programming.org/software/ConceptGCC/`, Jan. 2008.

[12] D. Gregor. Type-soundness and optimization in the concepts proposal. Technical Report N2576=08-0086, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Mar. 2008.

[13] D. Gregor and J. Siek. Implementing concepts. Technical Report N1848=05-0108, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Aug. 2005.

[14] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 291–310, Oct. 2006.

[15] D. Gregor, B. Stroustrup, J. Widman, and J. Siek. Proposed wording for concepts (revision 5). Technical Report N2617=08-0127, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2008.

[16] R. Hinze and S. Peyton Jones. Derivable type classes. In G. Hutton, editor, *Proc. 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[17] P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *Proc. POPL'97: Principles of Programming Languages*, pages 470–482. ACM, 1997.

[18] M. Jazayeri, R. Loos, and D. Musser, editors. *Generic Programming: International Seminar, Dagstuhl Castle, Germany, 1998, Selected Papers*, volume 1766 of *LNCS*, 2000. Springer.

[19] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proc. Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 52–61, New York, NY, USA, 1993. ACM.

[20] M. P. Jones. Type classes with functional dependencies. In *Programming Languages and Systems*, volume 1782 of *LNCS*, pages 230–244. Springer, 2000.

[21] M. P. Jones. Dictionary-free overloading by partial evaluation. *LISP and Symbolic Computation*, 8(3):229–248, 1994.

[22] W. Kahl and J. Scheffczyk. Named instances for Haskell type classes. In R. Hinze, editor, *Proc. 2001 ACM SIGPLAN Haskell Workshop*, pages 77–99. Elsevier Science, 2001.

[23] O. Kiselyov and S. Peyton Jones. Choosing a type-class instance based on the context, Apr. 2008. URL `http://haskell.org/haskellwiki/GHC/AdvancedOverlap`.

[24] O. Kiselyov and C.-C. Shan. Lightweight static resources, for safe embedded and systems programming. In *Draft Proceedings of Trends in Functional Programming*. Seton Hall University, 2007.

[25] J. Meacham. Class alias proposal for Haskell, 2006. URL `http://repetae.net/recent/out/classalias.html`.

[26] N. Mitchell. Deriving generic functions by example. Technical report, Dept. of Computer Science, University of York, UK, 2007. Tech. Report YCS-2007-421.

[27] G. Munkby, A. P. Priesnitz, S. Schupp, and M. Zalewski. Scrap++: Scrap your boilerplate in C++. In *WGP'06: Proc. of the 2006 ACM SIGPLAN Workshop on Generic Programming*, pages 66–75. ACM, 2006.

[28] S. Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.

[29] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, 1997.

[30] R. Rivera, B. Dawes, and D. Abrahams. The boost C++ libraries, 2008. URL `http://www.boost.org/`.

[31] A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. Technical report, Utrecht University, 2008. Short version is in Proc. 2008 ACM SIGPLAN Haskell Symposium.

[32] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *ICFP'08: The 13th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2008.

[33] T. Sheard. Generic programming in Ωmega. In *Datatype-Generic Programming*, volume 4719 of *LNCS*, pages 258–284. Springer, 2007.

[34] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, Dec. 2002.

[35] J. G. Siek and A. Lumsdaine. Essential language support for generic programming. In *PLDI '05: Proc. ACM SIGPLAN 2005 conference on Programming language design and im-*

plementation, pages 73–84, New York, NY, USA, June 2005. ACM.

[36] A. A. Stepanov and M. Lee. The standard template library. Technical Report HPL-95-11(R.1), Hewlett Packard Laboratories, Palo Alto, CA, USA, Nov. 1995.

[37] M. Sulzmann and M. Wang. Modular generic programming with extensible superclasses. In *WGP '06: Proc. 2006 ACM SIGPLAN workshop on Generic programming*, pages 55–65, New York, NY, USA, 2006. ACM.

[38] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *TLDI '07: proc. 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, New York, NY, USA, 2007. ACM.

[39] M. Sulzmann, G. J. Duck, S. Peyton-Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 17(1):83–129, Jan. 2007.

[40] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, Nov. 2002.

[41] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proc. 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–76. ACM, 1989.

[42] J. Willcock, J. Järvi, A. Lumsdaine, and D. Musser. A formalization of concepts for generic programming. In *Concepts: a Linguistic Foundation of Generic Programming at Adobe Tech Summit*. Adobe Systems, Apr. 2004.

[43] M. Zalewski, A. P. Priesnitz, C. Ionescu, N. Botta, and S. Schupp. Multi-language library development: From Haskell type classes to C++ concepts. In *MPOOL 2007*, 2007.