JFP **20** (3 & 4): 271–302, 2010. © Cambridge University Press 2010
doi:10.1017/S095679681000016X First published online 29 September 2010

Generic programming with C++ concepts and Haskell type classes—a comparison

JEAN-PHILIPPE BERNARDY and PATRIK JANSSON
Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, SE-412 96 Göteborg, Sweden
(*e-mail:* {bernardy,patrikj}@chalmers.se)

MARCIN ZALEWSKI
Open Systems Lab, Indiana University, Lindley Hall 215, Bloomington, IN 47405, USA
(*e-mail:* zalewski@osl.iu.edu)

SIBYLLE SCHUPP
Institute for Software Systems, Hamburg University of Technology, Schwarzenbergstraße 95 (E), D-21073 Hamburg, Germany
(*e-mail:* schupp@tuhh.de)

Abstract

Earlier studies have introduced a list of high-level evaluation criteria to assess how well a language supports generic programming. Languages that meet all criteria include Haskell because of its type classes and C++ with the concept feature. We refine these criteria into a taxonomy that captures commonalities and 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.

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 (2007) noted. In the Haskell community, *datatype-genericity* is normally the most important one. However, in this paper, we are interested in what Gibbons calls *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.

This paper is an extended version of Bernardy *et al.* (2008) and similar comparative work has also been done earlier. An "extended comparative study of language

support for generic programming" (Garcia *et al.* 2007) 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 version of GHC (the Glasgow Haskell Compiler) provides new features for associated types and ConceptGCC (Gregor 2008a), the only C++ compiler that currently supports concepts, turns concepts from mere documentation artefacts to a full-fledged C++ language feature (Dos Reis & Stroustrup 2006). Hence, when we refer to Haskell we mean the language implemented in the 6.10 release of GHC, and when we refer to C++ we mean the C++ standard draft (Becker 2009), the last one to support concepts in this round of standardisation. Concepts have been recently voted out from the next version of C++, but with a clear intention of revisiting the decision in only a few years (Gregor 2009; Stroustrup 2009b). The discussion in this paper is still relevant despite the removal of concepts from the standard draft: one of the reasons for delay is the need to work out some details of concept semantics. Our detailed comparison of concepts with type classes is a useful tool in the future discussion.

Due to those developments, the deficiencies that the comparison made by Garcia *et al.* (2007) revealed—in particular on the part of C++—no longer exist: C++ now provides as much support for concepts, associated types, and retroactive modelling as Haskell does. If one were to apply the taxonomy of Garcia *et al.* (2007) again, Haskell and C++ would, thus, be indistinguishable as languages for generic-programming. 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 as our 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 five of the altogether eight evaluation criteria and discussed various workarounds, we reckon that today all but one of the criteria are met (separate compilation is the remaining "failing" criterion—see Section 6.3 for details). 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 1, are:

- a refined set of criteria for evaluation of property-based generic-programming language facilities;

Table 1. *Comparison criteria and their support in C++ and Haskell. The first column gives the section where the feature is discussed*

3 Concepts	3.1	*Multi-type concepts*	●	●	Multiple types can be simultaneously constrained.
	3.1	*Multiple constraints*	●	●	More than one constraint can be placed on a type parameter.
	3.2.1	Type-functions as parameters	●	●	Concepts accept type-level functions as parameters.
	3.2.2	Value-level parameters	◐	⇆	Concepts accept value-level parameters.
	3.2.3	Defaults for parameters	●	⇆	Default values can be given to concept parameters.
	3.2.4	Functional dependencies	⇆	●	Type parameters can be specified to be functionally related.
4 Modelling	4	*Retroactive modelling*	●	●	New modelling relationships can be added after a concept (or type) has been defined.
	4.1	Monomorphic modelling	●	●	The simplest modelling: monomorphic types declared to model a concept.
	4.1.1	Parametric modelling	●	●	Support for declaring parametric families of modellings.
	4.1.2	Overlapping modelling	●	●	A type can model a concept via two modellings or more.
	4.1.3	Free-form modelling	●	●	Free-form contexts and heads can be used in modellings.
	4.2	Default definitions	●	●	Associated entities can be given default definitions in concept definition.
	4.2.1	Structure-derived modelling	⇆	◐	Modellings can be generated by structure of type definition.
	4.2.2	Modelling lifting	⇆	●	Modellings can be generated by lifting through wrapper types.
	4.2.3	Modelling propagation	●	⇆	Modelling of a refining concept can define modellings for refined concepts.
	4.2.4	Implicit definitions	●	⇆	Default modellings, definition of associated entities by names/signatures.
	4.2.5	Automatic modelling	●	⇆	For a concept, modellings are generated automatically.
5 Constraints	5.1	Constraints inference	◐	◐	Constraints to generic algorithms can be deduced.
	5.2	*Associated types access*	●	●	Concepts can have types as associated entities.
	5.3	*Constraints on associated types*	●	●	Concepts may include constraints on associated types.
	5.3	Type-equality constraints	●	●	Type variables can be constrained to unify to the same type.
	5.4	Constraint aliases	◐	⇆	A mechanism for creating aliases for concept expressions is provided.
	5.4	*Type aliases*	●	●	A mechanism for creating aliases for type expressions is provided.
6 Generic Algorithms	6.1	*Implicit argument deduction*	●	●	The arguments for the type parameters of an algorithm can be deduced.
	6.1	Result-based overloading	○	●	Inference of the modelling to use can be based on the return type.
	6.2	Concept-based specialisation	●	○	Algorithms can be specialised on existence of modelling.
	6.3	Separate type-checking	◐	●	Generic functions can be type-checked independent of calls to them.
	6.3	*Separate compilation*	○	●	Generic functions can be compiled independent of calls to them.

Italics is used for features previously introduced (by Garcia *et al.*). Double arrows (⇆) denote a missing feature that could be ported from the other language. For terminology, see Table 2.

- refined answers to the questions posed by Garcia *et al.* (2007), 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++. Some of the improvements emerging from the comparison have not been suggested before, such as modelling propagation for Haskell (Section 4.2.3).

2 Background and terminology

2.1 Examples

To set the scene, the 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. To reduce confusion and to better show similarities and differences between C++ and Haskell, we use the terminology of Garcia (Garcia *et al.* 2007) uniformly for both languages (see Table 2).

In Figure 1, we show the correspondence between C++ and Haskell by means of an example so that readers can transpose knowledge of one syntax to the other. This example features every important construct: a concept, a modelling, a generic algorithm constrained by the concept, and an admissible instantiation of the algorithm. 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 *Hashable*. The subsequent modelling is abstracted 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 for more details). The body of the modelling states the values for the associated types and functions of the *Hashset* concept: the type *element* is defined to be 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 either expressed by replacing the usual `typename` keyword preceding a type parameter with the name of the constraining concept or by listing one or more constraints after the `requires` keyword. In the remainder of this paper, we mostly use the "predicate" syntax, but in some examples we either augment it or replace it with the `requires` syntax. 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++ `Equality-Comparable` concept as given in the ConceptGCC 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

```
// 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++

```
-- 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

Fig. 1. Example of the same generic code in C++ and Haskell, showcasing the syntactical correspondence, line by line, for every major feature. Due to the differences between idiomatic Haskell and C++ styles, this example is necessarily artificial. Figure 2 provides a more realistic example.

```
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++

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

(b) Haskell

Fig. 2. Example of similar concepts from the standard libraries of C++ and Haskell. Due to the difference in style between Haskell and C++, differences exist between the implementations. Figure 1 provides an example with close correspondence.

Table 2. *Terminology and keywords*

Term	Haskell	C++
Concept	Type class **`class`**	Concept **`Concept`**
Refinement	Subclass **`=>`**	Refinement **`:`**
Modelling	Instance **`instance`**	Concept map **`concept_map`**
Constraint	Context **`=>`**	Requires clause **`requires`**
Generic algorithm	Polymorphic function *(no keyword)*	Function template **`template`**

apples and oranges, but a more typical example would be comparing apples and references to apples with a modelling like $EqComp(Apple, Apple\&)$. Figure 2(b) shows the `Eq` concept from the Haskell prelude (Peyton Jones 2003). The concept *Eq* 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.

2.2 Historical background

On the surface, the motivation and the origins of Haskell types classes and C++ Concepts seem to differ: type classes were added to support overloading, while C++ Concepts were added to improve type checking. However, careful examination reveals that the underlying ideas are similar, if not identical.

C++ had overloading capabilities from its inception. The meaning of the operator +, for example, is potentially different for each combination of argument types. Overloading is commonly used in combination with templates, by parameterisation

of a template definition over types. The definition can then be instantiated at various actual types, and the effect is that each occurrence of the template argument is substituted by its actual value. In the presence of overloading, the net result is that the meaning of the various template instantiations are unrelated. Therefore, template-style parameterisation is a form of ad-hoc[1] polymorphism. The unpredictability of ad-hoc polymorphism, however, is in direct opposition to the intent of authors of generic libraries. When they write +, they have in mind an operation with the usual properties of addition. Instantiating a template with a type that provides a + operator that does not meet these properties would then fail to meet the programmer's expectations. To address this problem, authors of C++ generic libraries started to document more precisely which assumptions were made about each operation used in the template. Famously, this practice was institutionalised in the Standard Template Library (STL) (Stepanov & Lee 1995). In essence, concepts in STL serve as a specification, much in the style of algebraic specification languages, such as Tecton (Kapur & Musser 1992).

While such specifications are useful to make sense of generic programs, they cannot help the compiler as long as they remain external to the language. In particular, a template cannot be type checked. Therefore, language support for concepts was recently proposed (Dos Reis & Stroustrup 2006; Gregor *et al.* 2006; Becker 2009), based on the practice established by the STL and other generic libraries.

On the other hand, the designers of Haskell quickly realised that the addition of unconstrained overloading to a language with parametric polymorphism would cause issues in type checking, and therefore overloading was only introduced in the form of type classes, as it emerged in (Wadler & Blott 1989).

In summary, both Haskell type classes and C++ concepts arose from the need to *structure* ad-hoc polymorphism; therefore, one should not be surprised that they turned out very similar to each other.

2.3 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 C++ (Stepanov & Lee 1995; Austern 1998), which Garcia *et al.* (2007) mapped to Haskell. Table 2 summarises it, updated with the new terminology from C++.

A *concept* can be considered an abstract specification of a set of (tuples of) types. The *arity* of a concept is the size of the tuple—the number of type parameters. Below, when we write predicate, one should understand (n-ary) predicates (or equivalently n-ary relations) for any n. Semantically, a concept has two aspects:

1. It corresponds to a *predicate* over types. When a tuple of types satisfies this predicate, we say that it is a *model* for the concept. Intuitively, such a type meets the concept specification.

[1] Cardelli & Wegner (1985) give a detailed exposition of the various flavours of polymorphism.

2. It has a number of *associated entities*: values, functions, or types[2]. 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.

A monomorphic *modelling* specifies how the associated entities of a concept are defined for one particular n-tuple of types. A concept can be described by a set of modellings, or equivalently, by a function from (tuples of) types to implementations of the associated entities. A parametric modelling (see Section 4.1.1) specifies a whole family of monomorphic modellings at once.

Generic algorithms are traditionally parametric over the types they accept, and thus correspond to template functions in C++ and polymorphic functions in Haskell. Concepts can then be used to *constrain* the type parameters to those algorithms, and conversely make the associated entities available inside the algorithms. A generic algorithm can then be *instantiated*, that is, applied to concrete types (which must satisfy the constraints). The type of the generic algorithm is polymorphic; it becomes monomorphic when *instantiation* binds all type parameters. The algorithm can then only be used for the types it was instantiated with.

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 break them down along the terms introduced in Section 2.3. Table 1 summarises the 28 criteria and the rest of the paper goes through them in more detail; the subsection where a criterion is discussed is given in the first column of the table. 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, in the following order, 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 are 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 number of 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

[2] A fourth kind of associated entity are axioms (Becker 2009): laws that other entities must satisfy. Because neither language supports axioms, we choose to leave them out of the discussion.

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 types.

However, the differences in approach outlined in Section 2.2 have consequences on the particulars of the compiler checks. In order to discover the differences, we start by outlining how concepts are checked and used 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 simplification, 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. If a generic algorithm is called from within another generic algorithm with one or more type variables as arguments, the compiler uses *archetypes*, i.e., placeholder types generated in the outer generic algorithm, to check the call. Archetypes automatically fulfil all the constraints placed in the outer algorithm, effectively propagating outer algorithm's constraints to the checking of the inner call. Importantly, the actual selection of modellings is delayed until algorithms are instantiated with monomorphic types, so that compile-time algorithm selection can be performed based on the available modellings. 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, as in `almostFull<MyHashSet>(x)`.

We can summarise the differences as follows:

- Haskell can infer the constraints from the use of associated entities, while C++ just checks that the entities used are in scope given the constraints provided by the programmer. In other words, 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" (i.e., instance declarations) 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 discuss these extensions along with functional dependencies in this section.

3.2.1 Type-functions as parameters

Since version 1.3, Haskell offers "constructor classes" (Jones 1993). This means that concepts cannot only apply to types, but also to *type constructors*. This feature has proven very useful in practice to model concepts like `Functor` or `Monad`.

```
class Functor f where -- f :: * -> *
  fmap :: (a -> b) -> f a -> f b

instance Functor List where -- example: parametric List
  fmap f Nil         = Nil
  fmap f (Cons a as) = Cons (f a) (fmap f as)
```

C++ does not offer type-constructors as such (see Lincke *et al.* 2009, for examples of applications where such machinery is needed). One, perhaps the most obvious, way to mimic the functionality of type functions is to use templates. Given this convention, it is possible to translate the `Functor` concept directly from Haskell to C++ as follows, taking advantage of the possibility to parameterise a concept by a parameter ranging over templates:

```
concept Functor<template<typename> class F> {
  template<typename A, typename B>
  function1<F<A>, F<B>> fmap (function1<A,B>);
}
```

In this concept, functions are represented by the `function1` class template from the Boost libraries (Boost). The `fmap` operation takes a function from some `A` to some `B` (`a -> b` in Haskell) and returns a function from `F<A>` to `F<B>` (`f a -> f b` in Haskell). Then, a modelling for lists can be written as follows:

```
template<typename T>
class List {
  /* some implementation */
};

concept_map Functor<List> {
  template<typename A, typename B>
  function1<List<A>, List<B>> fmap(function1<A,B> f) {
    /* some implementation */
  }
}
```

Yet, there is a problem with this solution. Parameters ranging over templates are not used widely in C++ for technical reasons (Vandevoorde & Josuttis 2002). As a workaround the `Functor` concept can be rewritten using another concept, `TypeConstructor`, to encode type constructors:

```
concept TypeConstructor<typename F>  {
  template<typename T>
  class Rebind;
};

concept Functor<TypeConstructor F> {
  template<typename A, typename B>
  function1<F::Rebind<A>,F::Rebind<B>> fmap (function1<A,B>);
}
```

The `TypeConstructor` concept requires the modelling to contain a `Rebind` class template that explicitly represents type constructor application in Haskell. In the above example, `Rebind` is applied to `A` and `B`, corresponding to constructor applications `f a` and `f b` in the corresponding Haskell example. In such an encoding, the type `F` has a different "kind" than in the previous encoding, as `F` is no longer a template itself, but it is required that a template application `Rebind` be provided in modellings of the `Functor` concept. In a sense, the functionality of a type constructor is disconnected from the type itself, and is stated as an associated entity. Thus, concepts make it possible to encode constructor classes by requiring templates, such as `Rebind`, that correspond to constructor application.

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 from a generic programming point of view: it means that some type-functions cannot be made models of any concept. This example works:

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

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

```
instance Functor (Λ value -> Map value key) where ...
```

but Haskell does not support type-level abstraction, and indeed unification is only first-order, so this is invalid.

3.2.2 Value-level parameters

In C++, concepts can also be parameterised over constants 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 encoding the

structures of the value level at the type level (Kiselyov & Shan 2007). However, the syntax of expressions under the above encoding is not natural, and thus one risks producing unreadable programs by using it. Therefore, the ability to parameterise concepts by values would be useful to port to Haskell. However, work towards full-fledged support for type-level data has been underway for a while (Sheard 2007), and we feel that it is now about to bear its fruits in the form of the *Strathclyde Haskell Enhancement* (McBride 2010) — thus it makes little sense to port the feature from C++ at this point.

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.

The semantics of default parameters for concepts in C++ is simple: every time a parameter to a concept is omitted in a constraint, its default value is inserted. If the default value contains occurrences of other parameter names, these are substituted with the actual parameters provided.

This mechanism does not rely on any C++-specific feature; therefore, it could in principle be ported to Haskell, which does not currently provides any similar feature. In practice, some potential obstacles may arise:

- the syntax for constraints mimics that of function application: the class appears to be applied to its parameters to form a constraint. Therefore, a class missing a parameter might feel like a partially applied relation – which is a totally different meaning from that given if the missing parameters would be filled in with default values. A different syntax for application of default parameters could then be invented to avoid the problem.
- The substitution mechanism might multiply the number of type variables in a constraint in a way that is not apparent to the programmer. Because limiting the number of type-variable occurrences in constraints is essential to ensure termination of type-checking, as we discuss in Section 4.1.3, the errors stemming from careless usage of default parameters could be very confusing.
- Type classes with many parameters are rare in Haskell compared to C++.

3.2.4 Functional dependencies

Jones (2000) 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 were not present in the concept *Hashset*(*set*, *element*), the dependency would be implicit by the non-existence of some models.

Making the restriction explicit in 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 (Sulzmann *et al.* 2007b), functional dependencies remain a controversial feature: Chakravarty *et al.* (2005a) argues that associated types provide much of the functionality but cause less complexity in the type system.

Functional dependencies are not available in C++. In C++, constraints are not inferred, so the usefulness of the feature would be more limited, but it could be used to enhance checking of type equality constraints. Functional dependencies could be enforced in exactly the same way as in Haskell. We do not list an example of the possible syntax, because the example would just transliterate Haskell code in into C++ without any significant insight.

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 call modelling flexibility (Section 4.1) and modelling shortcuts (Section 4.2).

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 but abstract from the associated entities: $M_1 \equiv Eq(bool)$.

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 a *context*, or an 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 type 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, is that one wants a pretty printer 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 the 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 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 (during type inference or type checking). 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 may be only partially unified to concrete types: the difficulty is to ensure 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.* (1997). Because 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 (Sulzmann *et al.* 2007b). 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 look up 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, modellings take the form $\forall a_1 \ldots a_n.\ (C_{r_1}\ a_{r_1}, \ldots, C_{r_k}\ a_{r_k}) C_m\ (t\ a_1\ \ldots\ a_n)$, 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 occurrences (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 (Kiselyov & Shan 2007).

Finally, we note that C++ modellings may be given in completely free form. Yet, to provide any non-trivial definitions, at least one argument of the modelled concept must be specialised, either by being unified with a type-constructor (either a built-in type constructor such as pointer formation, or a user-defined template) 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: GHC disables these features by default, requiring the use of compiler flags to turn them on. While C++ gives almost full flexibility, knowledge of the tradeoffs can be useful to programmers, so that complexity and the ensuing costs are understood.

4.2 Modelling shortcuts

Specifying modellings using the above constructs can sometimes be quite repetitive and therefore both C++ and Haskell provide syntactic sugar to define them concisely. Beyond the ability to give default definitions to associated entities in the concept declaration, the mechanisms in these languages are quite different.

4.2.1 Structure-derived modelling

While concepts allows for case-by-case (ad-hoc) modelling, the modellings themselves are often datatype-generic (Gibbons 2007). Examples include serialisation functions, equality checking, etc. For some concepts, deriving the modelling from the structure of types is sensible as a default, but for some types the programmer wants to override with 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 modellings (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 (Jansson & Jeuring 1997; Hinze & Peyton Jones 2001). Notably, Template-Haskell (Sheard & Peyton Jones 2002) provides a generic mechanism that suffices to implement a customised derivation construct (Mitchell 2007). There are also around ten proposals for the design of a generic library of concepts supporting polytypic modelling in Haskell (see Rodriguez *et al.* 2008 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 given explicitly. 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.* (2006) 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++. In summary, we mark this feature with "⇆" (portable) in Table 1, because a compiler could support structure-derived modellings for concepts such as *EqComp* as easily as it already supports generation of definitions for operators such as ==.

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, as a structure with a single field. The following example shows such a wrapper for integers:

```
struct Age { int age; };
```

Because 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 a few aspects of the wrapped type: one would like to reuse most modellings, but they must be given an explicit definition. Being mainly composed of wrapping and unwrapping, such a definition is typically easy but tedious do write. Therefore, 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.

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

In the above example, the `deriving` clause stands for the following modelling declaration:

```
instance Hashable Age where
  hash (Age i) = hash i
```

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, so this feature is a natural candidate to port. A difficulty is that `newtype` has no direct equivalent in C++, so the feature must be adapted to work on single-field structures.

4.2.3 Modelling propagation

C++ concepts can include associated entities of other concepts in two different ways, either through refinement or through nested constraints. The two mechanisms differ syntactically. Refinement is specified after the concept name, separated by a colon, but before the concept body. Nested constraints are given in the body of a concept and are preceded by the `requires` keyword. Semantically, the two mechanisms differ in two respects. First, they differ in how modellings are provided, which is the subject of this section. Second, the two mechanisms differ in how associated entities participate in name lookup within a concept: associated entities of refined concepts are included and associated entities of nested constraints are not.

The `Hashset` concept from Figure 1 uses both mechanisms, refining the `HasEmpty` concept and including a nested constraint for the `Hashable` concept:

```
concept Hashset<typename X> : HasEmpty<X> {
  typename element;
  requires Hashable<element>;
  int size(X);
}
```

When a modelling for `Hashset` is provided, for example, 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:

```
// must be given before modelling for Hashset
concept_map Hashable<int> { /* ... */ }

// The compiler verifies that modelling Hashable<int> is already defined
concept_map Hashset<int> {
  /* Hashset<int> definitions  */
  /* HasEmpty<int> definitions */
  // Compiler generates a modelling for HasEmpty<int> from definitions
  // and the surrounding scope
}
```

The programmer may provide definitions for the associated entities of the `HasEmpty` concept in the modelling declaration for `Hashset`; these definitions are then used to generate, or propagate, a modelling for `HasEmpty` when necessary. Because modellings for `HasEmpty` may be defined before modellings for `Hashset`, the semantics of C++ concepts gives *compatibility* rules by which definitions in the existing modellings of `HasEmpty` are checked for any possible conflicts with definitions in modellings of `Hashset`. The support for modelling propagation does not affect the power of concepts but it greatly simplifies use of concepts in practice—in large libraries refinements are often more numerous than in our simple example, and modelling propagation decreases the number of modellings that have to be written out by a programmer. 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 practice as it has for concepts in C++.

A concrete application would be a proposed refactoring of the numeric class in Haskell:

```
class (Eq a, Show a) => Num a  where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a
    fromInteger    :: Integer -> a
```

It has been suggested many times that the class should be split in (at least) two components, for example the following:

```
class Additive a where
    (+), (-)       :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a
    fromInteger    :: Integer -> a

class Multiplicative a where
    (*)            :: a -> a -> a
```

Then `Num` could be defined as follows:

```
class (Eq a, Show a, Additive a, Multiplicative a) => Num a  where
  -- no methods
```

While such a finer-grained hierarchy is arguably better, it is not backward compatible without modelling propagation. Indeed, a modelling such as

```
instance Num Bool where
  (+) = (||)
  (*) = (&&)
  {- etc. -}
```

would become invalid: `Num Bool` has no associated method, and no modelling for `Additive Bool` or `Multiplicative Bool` would be provided. What is needed is exactly the equivalent of C++'s feature: the above instance declaration should be translated into an `Additive` and a `Multiplicative` instance. In general, in an instance declaration, any set of methods belonging to a superclass should be translated to an instance declaration for the superclass.

Other, similar solutions to the problem have been proposed before. Notably, Meacham (2006) proposes *class aliases*. In that context, `Num` would be defined as

```
class alias Num a = (Additive a, Multiplicative a)
```

Due to the informal character of the proposal, it is difficult to compare class aliases to modelling propagation precisely. Two differences can be identified:

1. a class alias may define default methods for the aliased classes; and
2. there is no restriction on the classes that are aliased.

Orchard & Schrijvers (2010) note that some class aliases appear problematic:

```
class alias Eq' a b = (Eq a, Eq b)
```

Instances of `Eq'` should implement two equality operations, but the type instance to which each belongs may be indistinguishable in some cases. The same problem seems to arise with modelling propagation. We observe that it is much milder however. Consider:

```
class (Eq a, Eq b) => Eq' a b

instance Eq' Bool Int where
  x == y =  {- Primitive comparison on booleans -}
```

Presented with the above instance, the compiler will note that the method `(==)` does not belong to `Eq'`, and searches for superclasses where it can fit, based on the method name. In this case, both `Eq Int` and `Eq Bool` fit. Therefore, the following instances are created. Because the first one is type-incorrect, the code is rejected.

```
instance Eq Int where
  x == y =  {- Primitive comparison on booleans -}

instance Eq Bool where
  x == y =  {- Primitive comparison on booleans -}
```

In summary, in certain pathological cases, modelling propagation cannot be used to shorten definitions, but this restriction does not compromise its usefulness when the superclasses have nonoverlapping method names.

4.2.4 Implicit definitions

In C++, a modelling 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. We give a simple example based on the concept from Figure 2 where `!=` has a default implementation in terms of `==`:

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

The modelling *EqComp*(*int*) leaves all definitions implicit. The definition of `==` is deduced by the compiler, because `int` has an operator `==` with matching type. Then `!=` can use its default implementation, forwarding to `==`. The actual rules for implicit definitions are much more complex to allow for optimisations (Gregor 2008b)—our discussion is meant to give a basic idea of how the feature works.

In Haskell, direct porting of implicit definitions as performed in C++ is not realistic because, if a class is in scope, the functions that match the name and types of the methods of the class are coming from the class itself. Using the class methods as a definition for the instance method would effectively create a circular definition, which one would like to avoid. The mechanism could be adapted, though, if definitions could be taken from a *different* scope. Such a feature could be useful to retrofit old code to a new class. For example, imagine that a set data structure is defined in an existing module `OldSet`. It is likely that the names and types used in that module would largely match that of a new class, say the `Hashset` concept used as an example earlier. In that case, one can state that `OldSet` is an instance of `Hashset` as follows:

```
instance Hashset OldSet.Set where
  type Element OldSet = OldSet.Element
  size = OldSet.size
```

Because it is typical for a class to have many methods, a modelling declaration such as the above can be quite tedious to write. It would be more convenient to have the compiler pick definitions automatically from a specified scope, maybe using the following syntax:

```
instance Hashset OldSet.Set where
  import OldSet -- picking all definitions from the OldSet module.
```

The situation that we describe above occurs in practice: the module `Data.Edison.Assoc.AssocList` includes several modellings with up to 38 such trivial method declarations.

4.2.5 Automatic modelling

To further alleviate the burden of defining modellings and to ease retrofitting of existing types, C++ allows programmers 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 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> { }
```

Because `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 disagreement on whether non-auto or, using an equivalent term, explicit concepts should be the default or whether they should even be allowed at all has been an important factor in rejecting concepts from the next C++ standard (Gregor 2009; Stroustrup 2009b). Some committee members argued that most, if not all, modelling declarations would be such that the compiler could resolve them automatically and that forcing programmers to write "empty" modellings (see Section 4.2.4) would be a roadblock for many non-expert C++ programmers. Yet, explicit modellings are necessary to resolve cases where concepts do not differ or differ only minimally in the required syntax, but are considerably different in the required semantics. An example of such a situation are the `ForwardIterator` and the `InputIterator` concepts where `ForwardIterator` refines `InputIterator` without providing any new associated entities but only adds semantic requirements such as multiple-pass iteration. Accidental similarities between concepts can occur as well, but the proponents of automatic-only modelling argued that such cases are extremely rare. Stroustrup (2009a) proposed that automatic modelling should be the only kind and that only in cases such as the distinction between forward and input iterators a special kind of explicit refinement should be required. Explicit

refinement would enable non-automatic modelling only where it is necessary to resolve ambiguities. To summarise, there were two positions in the C++ committee during deliberations on concepts: one that automatic modelling should be the only kind except for cases where non-automatic modelling is necessary, and another that non-automatic modelling should be predominant with automatic modelling used only for ubiquitous concepts.

Can the experience of the Haskell community, which has been using concepts for a while, bring any useful information to the debate? There is no form of automatic modelling in Haskell: programmers must supply all modelling declarations explicitly. This has not proven a problem in practice: programmers are happy to type a few extra words per type declaration in order to be sure that each modelling is intended. Therefore, we believe that explicit concepts should be the default.

What would a mechanism similar to `auto` mean in Haskell? Whenever the compiler finds that a constraint is not satisfied, it should try to generate a modelling for an `auto`-concept to plug the hole. While this on-the-fly instantiation makes sense on the surface, it raises the question of how to generate the bindings of such an instance. Taking matching functions which are in scope, as described in Section 4.2.4, is not applicable: matching identifiers are in scope, but they come from the class, and thus do not make for a sensible definition. A possible solution would be to rely on structure-derived modelling (Section 4.2.1) to generate an instance on the basis of the structure of the type. What this amounts to is to provide a datatype-generic default definition for the methods of the class (GHC implements the Haskell extension "Generics" based on Hinze & Peyton Jones 2001). Lots of research have already been devoted to implementing seamless cohabitation of datatype-based and property-based generic programming in Haskell (Rodriguez *et al.* 2008). Our comparison adds just one more motivating point.

4.3 Summary

Property-based generic programming gives great flexibility: types can 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. In this section, we refine this statement and give some intuition for why they are different.

First, we note that Haskell allows the programmer to omit the constraint specification, but only if the type signature is omitted entirely. In other words,

it is an "all-or-nothing" choice: either the programmer specifies type information completely, including constraints, or gives up control, completely relying on the inference mechanism[3]. Also, C++ supports a limited form of inference, called *requirement implication* (requirements correspond to constraints in our terminology), in *declarations* of generic algorithms (Becker 2009). 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 constraint `Hashable<T>` to `g`. Similarly, to make the return type of `h` well formed in the following example:

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

the compiler adds the constraint `Returnable<Hashset<T>::element>` to `h` and similarly the use of a reference to `T` adds the constraint `ReferentType<T>`. Both `Returnable` and `ReferentType` are standard-library, compiler-supported concepts that correspond to basic properties of C++ types. Compiler-supported concepts are unusual in the sense that they do not list any associated entities explicitly, but they indicate that a certain language construct can be used with a given type. The `Returnable` concept, for example, signifies that a value of the modelling type can be returned from a function, but there is no way provided to express such requirement explicitly as an associated entity, and, consequently, the body of the `Returnable` concept is left empty. The standard library contains 23 such compiler-supported concepts to provide an interface between the concept system and the existing C++ type system (Becker 2009).

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 standard function overloading. 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 constraints of top of this behaviour would be awkward. Gottschling (2008) proposed constraint inference for C++ based on explicit inference declarations provided by the programmer, but his proposal was never fully integrated into the concepts proposal, leaving the idea only partially tested.

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

[3] There is a trick to guide the inference mechanism using dummy definitions—see `http://okmij.org/ftp/Haskell/partial-signatures.lhs`

5.2 Associated types

It was previously noted by Garcia *et al.* (2007) that one can encode associated types in Haskell with functional dependencies, but that such an encoding can be cumbersome. Partly in reaction to that observation, Chakravarty *et al.* (2005b) proposed an extension to allow associated data types, and not long after an extension supporting the full power of associated types (Chakravarty *et al.* 2005a), which was finally implemented by Schrijvers *et al.* (2008). In contrast, C++ concepts have included associated types from their inception, although concepts are not standard C++ yet.

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 of the concept.

An associated type can be seen as a function defined by pattern matching on the type level. Each modelling provides one equation explaining how a particular "type pattern" maps to a resulting type. The fact that modellings can be provided separately from the concept and the type means that these type functions are *open functions*—they can be extended with new equations in other modules. Haskell allows such type level functions (called type families) to be defined without defining a concept (type class) explicitly. As an example, the associated type `Element` from Figure 1 could be separated out from the *Hashset* concept:

```
1 type family   Element x
2 type instance Element (IntMap (List k)) = k
```

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 the elements of the hash set can be converted to strings themselves.

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

In the above type signature, a constraint is placed on the associated element type of `s`, `Element s`. In C++, associated types can be similarly accessed and constrained:

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

In particular, type-equality constraints on associated types are useful. For instance, to make sure that two hash sets have the same type of element, one can write the following constraint in Haskell.

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

Associated types are accessed as in the previous example, and type equality is required using the ~ syntax (think of ~ as a two-parameter concept written as an infix operator between its two type arguments). In C++, type equality is expressed using a compiler-supported concept `SameType`, as shown in the following declaration.

```
template<Hashset S1, Hashset S2>
requires std::SameType<Hashset<S1>::element,
                       Hashset<S2>::element>
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 (Sulzmann *et al.* 2007a). Schrijvers *et al.* (2008) 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. The implementation design for checking of type equality constraints in C++ has been discussed by Gregor & Siek (2005).

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.). This means that the semantics of `SameType` are similar to those of Haskell's type-equality constraints.

5.4 Constraint aliases

In Garcia *et al.* (2007) type aliases are recognised as 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 constraints (which are concept-level expressions).

As we have seen in Section 4.2.3, modelling propagation provides this feature to some extent. If, for example, we define a concept *Numeric* refining *Additive* and *Multiplicative* then *Numeric*(a) can be used in constraints in place of *Additive*(a), *Multiplicative*(a). However, this ability breaks down when part of the expression contains equality constraints or associated types. Indeed, such constructs are not allowed in the head of a modelling declaration. Orchard & Schrijvers (2010) give the example of the Monadic Constraint Programming framework (Schrijvers *et al.* 2009), which contains functions with complex constraints, such as the following:

```
eval :: (Solver s, Queue q, Transformer t,
         Elem q ~ (Label s, Tree s a, TreeState t),
         ForSolver t ~ s)  =>  ...
```

While the concept triple `(Solver s, Queue q, Transformer t)` could be abstracted as a refined concept and the type triple `(Label s, Tree s a, TreeState t)` could be abstracted as a type synonym, the resulting constraint would still be awkward to manipulate. There is a clear need for a separate constraint aliasing feature.

Sadly, neither Haskell nor C++ currently provides such a feature. The need has been recognised though: Orchard & Schrijvers (2010) propose to extend Haskell with constraint aliases.

While no top-level constraint aliasing is available in C++, local constraint aliasing has been proposed for C++ (Brown *et al.* 2008). Constraints in C++ have to be written out explicitly and, when associated types are constrained, parts of them are often repeated within a single constraint expression. To reduce repetition, a part of a constraint can be named for later use in the expression. For example, the constraints of an STL algorithm `std::inner_product` repeat a single constraint (see the use of `Multiplicable`) three times:

```
template<InputIterator Iter1, InputIterator Iter2, typename T>
requires Multiplicable<Iter1::reference, Iter2::reference> &&
  Addable<T, Multiplicable<Iter1::reference, Iter2::reference>::
    result_type> &&
  Assignable<T, Addable<T, Multiplicable<Iter1::reference,
    Iter2::reference>::result_type>::result_type>
T
inner_product(Iter1 first1, Iter1 last1, Iter2 first2, T init);
```

Using the proposed constraint aliases, particular constraints can be named and reused:

```
template<InputIterator Iter1, InputIterator Iter2, typename T>
requires mult = Multiplicable<Iter1::reference, Iter2:reference> &&
  add = Addable<T, mult::result_type> &&
  Assignable<T, add::result_type>
T
inner_product( Iter1 first1, Iter1 last1, Iter2 first2, T init );
```

Thorough usage of concept-based programming naturally results in complex constraints. Therefore, we believe that both top-level and local constraint aliases should be integrated to both Haskell and C++.

6 Generic 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 practice: 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 (Kiselyov & Peyton Jones 2008).

```
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.

The above modelling is possible in C++ as parametric modellings can be overloaded on constraints. However, it is more natural to overload a generic print algorithm without a `Print` concept:

```
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. In Haskell, separate type checking always guarantees safe instantiation of generic algorithms; C++, on the other hand, allows exceptions to separate type checking safety in the interest of generating optimal code. In the current concepts proposal, safety can be broken by concept-based specialisation, discussed in the previous subsection, or by allowing overloading resolution during instantiation, for details see (Gregor 2008b). These breaches in separate type checking reflect the long-standing practice of choosing the most efficient implementation (Jazayeri *et al.* 2000) 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 (representations of) its type arguments at run time. This is realised by a dictionary-passing translation (Wadler & Blott 1989). Jones (1995) explored how to compile away the dictionaries to obtain faster code, and current GHC often specialises away the overhead of dictionary passing when the type is statically known.

A similar style of separate compilation could be possible in the context of C++ (Gregor & Siek 2005; Gregor *et al.* 2006), but it would require serious changes to the language, such as adding virtual tables (dictionaries) for template parameters. Furthermore, concept-based specialisation and template specialisation would have to be restricted to cases where no ambiguity is ever possible. The cost of run-time

mechanisms is not compatible with the efficiency goals set out for templates. In summary, while separate compilation is possible, it would require changes that are unacceptable to the C++ community.

7 Related work

There is a considerable body of literature that analyses concepts, and their implementation in various languages, but to our knowledge only our earlier work (Zalewski *et al.* 2007; Bernardy *et al.* 2008) compares C++ and Haskell directly. We have first considered how to translate Haskell type signatures with class constraints to C++ concept-constrained templates. The purpose of such translation was to aid a development model where a prototype was first developed in Haskell and then translated to a C++ implementation. Our subsequent work was the direct predecessor of this paper. Here, we extend our previous investigation, in particular by substantiating the claims we made about portability of features. Besides, we include more code and examples, and we update our comparison for the recent developments in both Haskell and C++.

Peyton Jones *et al.* (1997) explored the tradeoffs 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 extensions of Haskell type classes, and some of the results need to be reinterpreted to make them accessible to a larger community.

Willcock *et al.* (2004) 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 in Garcia *et al.* (2007), the version of C++ that they considered did not include language support for concepts.

Siek & Lumsdaine (2005) 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.* (2007) or by us in the current paper, 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 (Kahl & Scheffczyk 2001). We have left a more thorough comparison of this feature to future 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.* (2007) with respect to the level of support for concepts. More specifically, our criteria capture differences in the way concept parameters, modelling definitions, constraints, and generic algorithms are treated in the two languages.

As Garcia *et al.* (2007) identified previously, 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 Sections 5.1 and 6.1.

Our study also points towards possible improvements both in Haskell and C++. On the Haskell side, we mainly add weight behind existing proposals: such as constraint aliases (Orchard & Schrijvers 2010), type-level data (McBride 2010), and data-type generic programming (Rodriguez *et al.* 2008). Additionally, the evolution of existing type-class hierarchies would be made much easier by porting modelling propagation (Section 4.2.3) from C++, and this does not seem to have been proposed before. Finally, implicit modelling definitions (Section 4.2.4) may become a useful feature as retrofitting existing code to a growing type-class hierarchy becomes also more common. On the C++ side, the imperatives of backward compatibility and performance consideration complicate porting features from Haskell. For example, separate compilation would significantly impact the runtime cost of concepts. However, we have identified some low hanging fruits: automatic lifting of modelling through wrapper types (Section 4.2.2) and structure-derived modelling (Section 4.2.1).

Our point-by-point comparison of Haskell and C++ features can be used as a tutorial for either type classes or concepts by a programmer coming from the other background. Therefore, we see our comparison not as much as a "benchmark" of support for generic programming, but rather as a bridge between two communities that consider generic programming as important.

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. Finding orthogonal comparison criteria will help to guide the design of languages with orthogonal generic programming features.

Out of our 28 criteria, summarised in Table 1, 16 are equally supported in both languages, and only three come from fundamental differences in approach. So, we can safely conclude as we started — C++ concepts and Haskell type classes *are* very similar.

Acknowledgements

We thank Andreas Priesnitz and Gustav Munkby for fruitful discussions on the topic. The anonymous reviewers were very helpful in improving both the presentation and the content of the paper.

References

Austern, M. H. (1998) *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc..

Becker, P. (ed) (June 2009) *Working Draft, Standard for Programming Language C++* . Number N2914=09-0104 in JTC1/SC22/WG21 - C++. ISO/IEC.

Bernardy, J., Jansson, P., Zalewski, M., Schupp, S. & Priesnitz, A. (2008) A comparison of C++ concepts and Haskell type classes. In *Proceeding of the ACM SIGPLAN Workshop on Generic Programming (WGP 2008)*. ACM, pp. 37–48.

Boost. (2009). The Boost initiative for free peer-reviewed portable C++ source libraries. Available at: `http://www.boost.org`.

Brown, W., Jefferson, C., Meredith, A. & Widman, J. (2008) *Named Requirements for C++ Concepts*. Technical Report N2581=08-0091, ISO/IEC JTC1/SC22/WG21 - C++.

Cardelli, L. & Wegner, P. (1985) On understanding types, data abstraction, and polymorphism, *ACM Comput. Surv.*, 17 (4): 471–523. ISSN 0360-0300.

Chakravarty, M. M. T., Keller, G. & Peyton Jones, S. (September 2005a) Associated type synonyms, *SIGPLAN Not.*, 40 (9): 241–253.

Chakravarty, M. M. T., Keller, G., Peyton Jones, S. & Marlow, S. (2005b) Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, pp. 1–13.

Dos Reis, G. & Stroustrup, B. (2006) Specifying C++ concepts. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, pp. 295–308.

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

Gibbons, J. (2007) Datatype-generic programming. In *Spring School on Datatype-Generic Programming*, volume 4719 of *LNCS*. Springer, pp. 1–71.

Gottschling, P. (2008) *Concept Implication and Requirement Propagation.* Technical Report N2646=08-0156, ISO/IEC JTC1/SC22/WG21 - C++.

Gregor, D. (January 2008a) ConceptGCC — a prototype compiler for C++ concepts [online] Available at: `http://www.generic-programming.org/software/ConceptGCC/`.

Gregor, D. (2008b) *Type-Soundness and Optimization in the Concepts Proposal.* Technical Report N2576=08-0086, ISO/IEC JTC1/SC22/WG21 - C++.

Gregor, D. (August 2009) What happened in Frankfurt? C++-Next, The next generation of C++ [online]. Available at: `http://cpp-next.com/archive/2009/08/what-happened-in-frankfurt/`.

Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G. & Lumsdaine, A. (2006) Concepts: Linguistic support for generic programming in C++. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pp. 291–310.

Gregor, D. & Siek, J. (August 2005) *Implementing Concepts*. Technical Report N1848=05-0108, ISO/IEC JTC1/SC22/WG21 - C++.

Hinze, R. & Peyton Jones, S. (2001) Derivable type classes. In *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, Hutton, G. (ed), vol. 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science.

Jansson, P. & Jeuring, J. (1997) PolyP — a polytypic programming language extension. In *Proceedings of the Principles of Programming Languages (POPL 1997:)*. ACM Press, pp. 470–482.

Jazayeri, M., Loos, R. & Musser, D., (eds) (2000) In *Generic Programming: International Seminar, Dagstuhl Castle, Germany, 1998, Selected Papers*, vol. 1766 of . Springer.

Jones, M. P. (1993) A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA)*. ACM, pp. 52–61.

Jones, M. P. (1995) Dictionary-free overloading by partial evaluation. *LISP Symb. Comput.*, 8 (3):229–248.

Jones, M. P. (2000) Type classes with functional dependencies. In *Programming Languages and Systems*, vol. 1782 of *LNCS*. Springer, pp. 230–244.

Kahl, W. & Scheffczyk, J. (2001) Named instances for Haskell type classes. In *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, Hinze, R. (ed), Elsevier Science, pp. 77–99.

Kapur, D. & Musser, D. (1992) *Tecton: A Framework for Specifying and Verifying Generic System Components*. Technical Report, Rensselaer Polytechnic Institute [online]. Available at: URL `http://www.cs.rpi.edu/ musser/gp/tecton/tpcd-tecton.ps`.

Kiselyov, O. & Peyton Jones, S. (April 2008) Choosing a type-class instance based on the context [online]. Available at: `http://haskell.org/haskellwiki/GHC/AdvancedOverlap`.

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

Lincke, D., Jansson, P., Zalewski, M. & Ionescu, C. (July 2009) Generic libraries in C++ with concepts from High-Level domain descriptions in Haskell. In *IFIP TC 2 Working Conference on Domain-Specific Languages*, pp. 236–261.

McBride, C. (2010) She's faking it [online]. Available at: `http://personal.cis.strath.ac.uk/ conor/pub/she/faking.html`.

Meacham, J. (2006) Class alias proposal for Haskell [online]. Available at: `http://repetae.net/recent/out/classalias.html`.

Mitchell, N. (2007) *Deriving Generic Functions by Example*. Technical Report, Dept. of Computer Science, University of York, UK. Tech. Report YCS-2007-421.

Munkby, G., Priesnitz, A., Schupp, S. & Zalewski, M. Scrap++: Scrap your boilerplate in C++. (2010) In *Proceedings of the 2006 ACM SIGPLAN Workshop on Generic Programming (WGP 2006)*. ACM Press, pp. 66–75.

Orchard, D. & Schrijvers, T. (2010) Haskell type constraints unleashed. In *Proceedings of the 2010 International Symposium on Functional and Logic Programming (FLOPS 2010:)*. Springer.

Peyton Jones, S. (2003) *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press.

Peyton Jones, S., Jones, M. & Meijer, E. (1997) Type classes: an exploration of the design space. In *Haskell Workshop* [online]. Available at: `http://research.microsoft.com/en-us/um/people/simonpj/papers/type-class-design-space/`.

Rodriguez, A., Jeuring, J., Jansson, P., Gerdes, A., Kiselyov, O. & Oliveira, B. C. d. S. (2008) Comparing libraries for generic programming in Haskell. In *Haskell 2008*. ACM, pp. 111–122.

Schrijvers, T., Peyton Jones, S., Chakravarty, M. & Sulzmann, M. (2008) Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (ICFP 2008). ACM Press.

Schrijvers, T., Stuckey, P. & Wadler, P. (2009) Monadic constraint programming, *J. Funct. Program.*, 19 (6): 1–35.

Sheard, T. Generic Programming in Ωmega. (2007) In *Datatype-Generic Programming*, vol. 4719 of *LNCS*. Springer, pp. 258–284.

Sheard, T. & Peyton Jones, S. (December 2002) Template meta-programming for Haskell. *SIGPLAN Not.*, 37 (12): 60–75.

Siek, J. G. & Lumsdaine, A. (June 2005) Essential language support for generic programming. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005)*. ACM Press.

Stepanov, A. A. & Lee, M. (November 1995) *The Standard Template Library*. Technical Report HPL-95-11(R.1), Hewlett Packard Laboratories, Palo Alto, CA, USA.

Stroustrup, B. (June 2009a) *Simplifying the Use of Concepts*. Technical Report N2906=09-0096, ISO/IEC JTC1/SC22/WG21 - C++.

Stroustrup, B. (July 2009b) The C++0x "Remove Concepts" Decision. *Dr. Dobb's*.

Sulzmann, M., Chakravarty, M. M. T., Peyton Jones, S. & Donnelly, K. (2007a) System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international Workshop on Types in Languages Design and Implementation (TLDI 2007)*. ACM Press, pages 53–66.

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

Vandevoorde, D. & Josuttis, N. M. (November 2002) *C++ Templates: The Complete Guide*. Addison-Wesley Professional.

Wadler, P. & Blott, S. (1989) How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1989:)*. ACM Press, pp. 60–76.

Willcock, J., Järvi, J., Lumsdaine, A. & Musser, D. (April 2004) A formalization of concepts for generic programming. In *Concepts: a Linguistic Foundation of Generic Programming at Adobe Tech Summit*. Adobe Systems.

Zalewski, M., Priesnitz, A., Ionescu, C., Botta, N. & Schupp, S. (2007) Multi-language library development: From Haskell type classes to C++ concepts. In *Multiparadigm Programming with Object-Oriented Languages: MPOOL 2007*.