Software Extension and Integration with Type Classes

Ralf Lämmel
Microsoft Corp., Data Programmability Team, USA

Klaus Ostermann
Darmstadt University of Technology, Germany

Abstract
The abilities to extend a software module and to integrate a software module into an existing software system without changing existing source code are fundamental challenges in software engineering and programming-language design. We reconsider these challenges at the level of language expressiveness, by using the language concept of type classes, as it is available in the functional programming language Haskell. A detailed comparison with related work shows that type classes provide a powerful framework in which solutions to known software extension and integration problems can be provided. We also pinpoint several limitations of type classes in this context.

Categories and Subject Descriptors D.2.13 [SOFTWARE ENGINEERING]: Reusable Software

General Terms Design, Languages

Keywords Software extension, Software integration, Haskell, Type classes, Object adapter, Tyranny of the dominant decomposition, Expression problem, Multiple dispatch, Family Polymorphism, Framework integration

1. Introduction
Software extension and integration are fundamental and well-known challenges in software engineering and programming-language design. We refer to software extension as the act of extending a software module without modifying it, and to software integration as the act of integrating a software module into an existing software system. Extension and integration are related to modularity; only those properties of a system that are implemented in a modular way can be refined or replaced without modifying existing source code — this is a premise that dates back to Parnas [48].

Although the ability to extend software used to be a prime motivation for class inheritance when the OO paradigm was conceived, it is well understood by now that both class and interface inheritance are not flexible enough to express several kinds of extensions. There exist formulations of extension or integration problems such as the expression problem [58, 60, 57, 36], the framework integration problem [37], the problem of independent extensibility [54, 55], the tyranny of the dominant decomposition [20, 56], scattering and tangling [29], or the component integration problem [23, 40]. These documented problems have been used in the past to illustrate limitations of OO languages and to motivate various language extensions or encoding schemes.

In the present paper, we revisit the challenges for software extension and integration while putting to work the language concept of type classes [27, 59, 18], as available in the functional programming language Haskell. It turns out that type classes provide a powerful modularity mechanism that can be used to address several of the aforementioned extension and integration problems. It is worth stressing that our use of type classes directly targets software extension and integration; we do not detour through encodings of OO features such as mutable objects or object-type inheritance in Haskell — even though type classes are known to be useful in this context, too [31].

Contributions (i) We provide a new look on type classes from the perspective of software extension and integration. There is little or no related work that furnishes such a view; a noteworthy exception is Siek et al.’s work on generic programming [17, 52, 51]. (ii) We explain the advanced modularity that is provided by type classes, (iii) We describe concrete, type-class-based solutions for some well-known modularity problems. (iv) As a byproduct, we identify some aspects of type classes that limit their usefulness for software extension and integration. (v) We reveal the relevance of Haskell (and its type classes) for expressiveness problems mainly studied in the OO research community.

Road-map Sec. 2 and Sec. 3 discuss various well-known extension and integration problems. Sec. 2 facilitates conventional single-parameter type classes, while Sec. 3 also employs multi-parameter type classes. Sec. 4 analyzes the characteristics of type classes at a more abstract level.

2. Single-parameter type classes at work
After a short introduction to type classes, we will look at well-known problems that require only basic Haskell 98-based type-class expressiveness dating back to 1988-89 [27, 59]. That is, we use single-parameter type classes with types as parameters. It is surprising to see (for these authors anyhow) how far we can get with such folklore expressiveness.

2.1 Introduction to type classes
Type classes were originally developed to deal with ad-hoc polymorphism such as operator overloading in a less ad-hoc manner [27, 59]. A type class defines a signature for a family of functions that every member of the type class must implement. For instance, the following type class, Eq, comprises one member function (==), which is meant to be a predicate for equality:

class Eq a where
  (==) :: a -> a -> Bool

This class is parameterized in a single type, a. A type becomes a member of a type class by an instance declaration that implements the functions declared in the type class. For instance, consider the following datatype for Peano naturals (0, successor of 0, ...):
Any use of 'overloaded' function symbols (such as (==)) in a program is subject to instance selection. The selection happens at compile time, if argument and result types are sufficiently instantiated; it may be deferred to run-time otherwise, e.g., by compilation to explicit dictionary passing style. Type classes of a certain form are part of the standardized Haskell-98 language [49]; several more elaborate type-class features have been in use for some years now [9, 24, 50, 25, 12], and several of them are likely to get into the emerging Haskell standard.

2.2 Object adapter

As a warm-up, we consider the ‘adapter problem’, i.e., the problem of adapting an object type so that it implements the interface that is required by a given client. This problem is addressed by the OO design pattern adapter. Fig. 1 shows the example used by Gamma et al. [16]. The class TextShape in Fig. 1 is an object adapter that wraps objects of type TextView and provides an implementation of the boundingBox method in terms of TextView’s protocol.

Fig. 2 shows the type-class-based solution of the adapter problem. The declaration data TextView declares an algebraic datatype that corresponds to the OO class TextView. The functions origin and size correspond to the methods of the TextView class. The interface Shape is transcribed to a type class in Haskell. The type class Shape comprises one method, boundingBox, whose first argument type is the type parameter of the class. The declaration instance Shape TextView makes TextView an instance of the Shape type class; an appropriate implementation of boundingBox is provided; it calls out to TextView-specific functions origin and size.

The function circumference (at the bottom of the figure) illustrates type-class-bounded polymorphism, which roughly corresponds to OO interface polymorphism. The function takes a parameter whose type must be an instance of Shape (denoted by the constraint Shape s in front of the => sign). Here is an expression that applies the polymorphic circumference function to a text view:

\[
\text{circumference} \, (\text{TextView} \, \text{"foo"}) = 0.0 \, 10 \, 10 \, 10
\]

**Assessment** The strength of the type-class-based solution is that it does not need any wrapper type such as TextShape in the OO approach. Instead, the adapter type TextView is equipped with a Shape implementation by means of the instance declaration. Wrapping is known to cause schizophrenia due to confusion between the identity of the wrapper and the wrapper. Also, when an adapter is aggregated by an object, then one cannot invoke the adapter interface directly; cf. Fig. 3.

The type-class-based approach comes with an arguable restriction: the adapter must not require data extension, i.e., it must be ‘stateless’. One may take the position that an adapter is not supposed to carry state. Then, this restriction is still a hint at potential limitations with regard to design patterns other than adapter.

2.3 Tyranny of the dominant decomposition

Our next problem is the one of the ‘tyranny of the dominant decomposition’ [56], which is essentially about the OO programming limitation that one has to choose a fixed decomposition of a system in the design of a class hierarchy, although other decompositions are appropriate, too. A standard example is the classification of trees and plants from the perspective of a lumberjack and a bird, respectively [56, 20]. A lumberjack would classify trees into categories like hard wood and soft wood trees; a bird is more interested in properties like whether a plant provides nectar or insects, hence his decomposition of the system would be very different.

Fig. 4 shows an encoding of the two independent hierarchies using type classes. The => operator in the class declaration, cf. class Plant p => NectarPlant, has the same purpose as in a function declaration: it restricts the set of admitted instances of NectarPlant to those types that are instances of Plant. This use of => is similar to OO interface inheritance.

Let us now consider a concrete tree, namely a cherry tree, which is a hardwood tree according to the lumberjack’s view, and a nectar plant according to the bird’s view. Fig. 5 shows a model of cherry trees as a standard algebraic datatype with properties such as its
The basic notion of closed algebraic datatypes. The existing datatypes, while the introduction of new data variants is not readily admits the introduction of new operations on exist-

basic notion of closed classes. Conversely, functional programs dimension (read as ‘new methods’) is not admitted by the mechanism of class inheritance; extensibility in the opera-

sion problem is the problem of achieving exten-

2.4 The expression problem

size; the two independent decompositions of the tree domain are described by corresponding type-class instances.

Assessment The type class solution is easy to extend in a modular way: Consider a third decomposition of the tree domain from the point of view of a lawyer. This view may classify trees according to applicable taxes or laws. No code has to be changed to add such an additional decomposition. The perspective of the lawyer can be specified in one module; the instance declaration, which maps the cherry tree to the perspective, can be specified in yet another module.

The type-class-based addition of a new perspective is surpris-

ingly convenient, when compared to the approaches that were designed to address the problem of the tyranny of the domi-

An open type class for expression forms

class Exp x

-- An open type class for expression forms
class Exp x

data Exp = Lit Int | Add Exp Exp

-- An operation for evaluation
eval :: Exp -> Int
eval (Lit i) = i
eval (Add l r) = eval l + eval r

-- Another operation – for printing
print :: Exp -> IO ()
print (Lit i) = putStrLn (show i)
print (Add l r) = do print l; putStrLn “ + ”; print r

Figure 6. Extensibility in the operation dimension

Concrete expression forms
data Lit = Lit Int
data (Exp x, Exp y) => Add x y = Add x y
instance Exp Lit
instance (Exp x, Exp y) => Exp (Add x y)

-- An evaluation operation
class Exp x => Eval x where eval :: x -> Int
instance Eval Lit where eval (Lit i) = i
instance (Eval x, Eval y) => Eval (Add x y) where
  eval (Add x y) = eval x + eval y

Figure 7. Extensibility both of data and operations

expression problem and related language expressiveness have been studied in depth [58, 60, 57, 36].

Fig. 6 recalls the standard example for the expression problem; the figure actually shows the basic Haskell version that is not extensible in the data dimension. Fig. 7 illustrates the type-class- based recipe for extensibility in both of the dimensions data and operations; the recipe is easily summarized as follows:1

• Each data variant of an extensible datatype is modeled as a separate datatype which is polymorphic in the recursive occurrences of the extensible datatype; cf. the constructor Add for binary addition that carries two type parameter accordingly.

• The extensible, nominal union over all data variants is modeled as a type class; cf. the type class Exp that comprehends all expression forms. This type class also serves as a bound for the aforementioned type parameters.

• Each extensible operation on the extensible datatype is modeled as a type class that subclasses the type class of the extensible datatype on which to dispatch; cf. the type class Eval for expression evaluation with instances for Lit and Add.

Assessment Extensibility in the data dimension is straightforward: first add the designated datatype for the data variant, then add the variant to the union with an instance. Finally extend some or all operations to cover the new variant. Extensibility in the operation dimension is straightforward, too: add a new type class for the operation, then provide instances for some or all data variants in scope. Both extensibility dimensions are illustrated in Fig. 8.

The approach enables separate compilation; for example, all definitions in Fig. 7 can be compiled independently of those in

1 This recipe has been around for a while. In particular, the first author had posted this recipe to haskell.org and comp.compilers previously [33]. Also, see [35] for a more recent discussion.
Fig. 8. Extensions in the data and operation dimensions

Fig. 9. Non-extensible, value-level dispatch

Fig. 10. Extensible, type-level dispatch

3. Multi-parameter type classes at work

We have not fully exhausted all Haskell 98-based type-class expressiveness. In particular, we could have looked for extension and integration problems that require constructor classes [26]; i.e., type classes whose parameters are supposed to be type constructors. This topic may be an interesting future-work item.

In the present section, we have collected solutions to problems that take advantage of multi-parameter type classes [9, 24, 50]. Thereby, we go beyond the Haskell 98 standard, but multi-parameter type classes of some form are likely to make it into the emerging Haskell standard, and they are supported by the prominent Haskell implementations for several years now. For most of this section, we will restrict ourselves to the uncontroversial subset of the design space for multi-parameter type classes.

3.1 Multiple dispatch

Let us consider the problem of expressing case discriminations that involve multiple ‘objects’. Languages like Java and C# are limited in this respect (as of writing); they provide a single dispatch protocol that uses the receiver type of messages for the case discrimination based on modularization in OO methods. Generalizations of single dispatch for virtual methods have been proposed in the form of multiple dispatch or multi-methods [8, 10].

Let us consider a simple example adopted from [10] — an intersection method for shapes, where different combinations of shapes may admit different (more efficient) implementations of intersection method for shapes, where different combinations of shapes may admit different (more efficient) implementations of intersection method for shapes, where different combinations of shapes may admit different (more efficient) implementations of intersection method for shapes.
as a designated type (just as much as in OO). A multiple dispatch is captured by the method of a multi-parameter type class. Each relevant combination of shapes is modeled as a type-class instance. The instances in Fig. 10 directly encode the equations in Fig. 9.

**Assessment** The non-extensible, pattern-matching-based ‘solution’ provided us with strong static completeness checking, if we assume that exhaustiveness of pattern matching is checked. The type-class-based solution provides the same lazy completeness checking as we discussed for the expression problem. This time, one may argue that such laziness (i.e., checking coverage of actually attempted cases) is beneficial because it liberates us from the requirement to cover all combinations or to provide a polymorphic default as required by MultiJava.

On the design scale of multi-parameter type classes, there is the controversial concept of overlapping instances, which is nevertheless supported by the prominent Haskell implementations for several years now. We can use overlapping instances to provide a generic default — if this is considered useful for the (single or multiple) dispatch scenario at hand. Suppose we had an (‘inefficient’) default implementation of intersection for all kinds of shapes:

```haskell
instance (Shape x, Shape y) => Intersect x y where
  intersect s1 s2 = ... -- generic default for SxS
```

Further suppose that we have the following ‘efficient’ cases:

```haskell
instance Shape x => Intersect Rectangle x where ... -- RxS
instance Shape x => Intersect Rectangle y where ... -- SxR
```

The first instance gives a generic default implementation for intersection of two shapes; the second and third instance specify more special cases where the left or the right argument, respectively, are rectangles. The normal semantics of overlapping instances is such that the most specific instance is chosen. For instance, if we have a circle c and a rectangle r, then applications of `intersect` are dispatched as follows:

```
intersect c c = ... -- selects SxS instance
intersect c r = ... -- selects RxS instance
intersect r r = ... -- static error: ambiguous selection
```

The last case is reported as an error because both RxS and SxR match but none of them is more specific than the other. If we added an RxR instance, then the last `intersect` call would be valid since the RxE instance would count as the most-specific applicable instance.

In most multi-dispatch systems, a more conservative regime is applied. That is, potentially ambiguous method combinations are not admitted. The type-class-based solution is less conservative. One can provide instances with potential ambiguity. An error is raised only when an ambiguity is statically detected at a call site. A transcription of this powerful regime to OO languages would need to accommodate the non-trivial complication of OO subtyping.

### 3.2 Family polymorphism

Traditional subtype polymorphism cannot capture relations between several objects and their methods. For instance, it is not possible to specify that a method accepts a node and a vertex object while both objects need to belong to the same family such as a family for colored graphs or weighted graphs. As a remedy, the notion of family polymorphism [13] has been proposed with type-system realizations based on virtual classes and path-dependent types [15, 13].

Fig. 11 shows a type-class encoding of (a variant of) the graph example from [13]. The type class `Graph` is a multi-parameter type class with three type parameters g, v, and e standing for the graph family, the vertex type and the edge type, respectively. Since we want vertices to be comparable, we restrict v by `Eq v`. The annotation `g -> v e` is the declaration of a functional dependency [25],

```haskell
class Eq v => Graph g v e | g -> v e where
  vertices :: g -> v e -> (v,v)
  edges :: g -> v v -> [e]
otherVertex :: Graph g v e => g -> v -> e -> v
otherVertex g v e = if (v == v1) then v2 else
  if (v == v2) then v1 else ⊥
where (v1,v2) = vertices g e
```

**Figure 11.** A type class for graphs and a simple function on graphs

```haskell
instance Graph [[Int]] Int (Int,Int) where
  vertices (v1,v2) = (v1,v2)
  edges g v = map (\i -> (v,i)) (g !! v)
data Vertex = Vertex String [Edge] deriving Eq
data Edge = Edge Float Vertex Vertex deriving Eq
instance Graph [Vertex] Vertex Edge where
  vertices (g Edge _ vl v2) = (v1,v2)
  edges g (Vertex _ es) = es
```

**Figure 12.** Two different graph families

```haskell
graph1 = [[1],[0]] -- 1st family
graph2 = [v1,v2] where -- 2nd family
  v1 = Vertex "x" e1
  v2 = Vertex "y" e2
  e1 = Edge 0.5 v1 v2
e2 = Edge 0.7 v2 v1
test1 = otherVertex graph1 1 (0,1)
test2 = otherVertex graph2 v1 e1
  -- test3 = otherVertex graph2 v1 (0,1) -- static type error
```

**Figure 13.** Family polymorphism at work

which means that the graph type determines the vertex and edge types. The function `otherVertex` is an example for a (very simple) algorithm on graphs using the `Graph` type class.

Fig. 12 shows two different instances for the `Graph` class, one using integer lists, and one using algebraic datatypes. Fig. 13 illustrates the use of these two different graph families. The last line in the figure illustrates static typing; any attempt to confuse the families is rejected by the type checker.

For comparison. Fig. 14 shows an encoding of the same example in `CaesarJ` [41, 1], a Java-based language with virtual classes and path-dependent types. The same example could just as well be encoded in `gbeta` [13] or Scala [46]. In Fig. 14, `Edge` and `Vertex` are virtual classes, which means that they are class-valued properties of the enclosing instance of `Graph`. The class `ConcreteGraph1`, which is a subclass of `Graph` can refine the definition of `Edge` and `Vertex` with a concrete implementation.

**Assessment** Virtual classes rely on type parameterization by values, which is more flexible than type parameterization by types, as provided for type classes. For instance, the types `g1.Edge` and `g1.Vertex` depend on a first-class `value`, namely an object `g1`, whereas the edge and vertex type in the corresponding type class `Graph` depend on a `type`, namely the first type parameter of the class, as expressed through the functional dependency. For instance, we could have a dynamic selection of a graph based on a test:

\[\text{deriving Eq} \] for the algebraic datatypes `Vertex` and `Edge` means that the `Eq` instance should be derived canonically by the compiler. Haskell 98 provides special support for some type classes including `Eq`.\[\text{deriving Eq}\]
is important to notice that all nested classes in CaesarJ are virtual classes and hence

Figure 14. The graph example rephrased in CaesarJ [41, 1] — is

Figure 15. Framework example from [21, 42]

framework integration

mation can now be called with values of adapted application types,

framework integration

framework classes, the integration of a framework into an

is a virtual class. In superclass declarations or constructor invocations.

existing application contains hard links to the original application

This kind of mapping is non-invasive and very flex-

Assessment This kind of mapping is non-invasive and very flex-

Application

The integration of OO frameworks into an existing application

framework integration

framework integration

The integration of OO frameworks into an existing application

framework integration

framework integration

Figure 16. Type-class encoding of the framework in Fig. 15

data RegularPricing = RP

instance Pricing RegularPricing

Figure 17. Non-invasive integration of the framework in Fig. 16

Fig. 17 alludes to a binding of the framework to a particular

Framework

framework integration

framework integration

framework integration

framework integration

framework integration

framework integration

framework integration

framework integration

framework integration
4. Discussion

We are ready to draw some conclusions from the examples of the previous sections. In particular, we identify the particular strengths of type classes, when compared to basic or advanced OO expressiveness. Furthermore, we comment on some limitations of type classes in the context of software extension and integration.

4.1 Retroactive interface implementation

It is striking that single-parameter type classes are similar to interfaces in OO languages like Java and C#. The main difference between the two concepts is in the modularity and extensibility: implementation of an interface in an OO class must be designed into the class, hence it is not possible to separate the interface implementation from the OO class definition or to add an interface implementation to an existing class, which is probably the main reason for the ubiquitous usage of adapters in OO languages.

Type classes enable fine-grained modularity because an instance can be declared separately from both the class declaration and the type which is made an instance of the class. In OO terms, such a separation into three independent parts corresponds to retroactive interface implementation. One can achieve a similar degree of decomposition by using the OO design pattern strategy [16], in which case behavior (according to a given interface) is separated from objects. The main difference between strategies and type classes is that the former approach requires one to explicitly select and pass implementations — as if we were encoding the latter approach based on ‘explicit dictionary passing’.

Haskell’s type classes assume global scope for instances: an instance defined by any of a program’s modules will be visible (i.e., applicable) anywhere in the program; regardless of module hierarchy and module interfaces. While such global scope is mostly appropriate and often beneficial for retroactive interface implementation, local scoping may be useful at times, but it would also challenge coherence and the definition of principal types [59, 30].

The type-class-based solutions of Sec. 2 exploit the retroactive capability. In OO terms, one would need to be able to ‘re-open’ a class in a separate module so that we can add an implementation for an interface referred to by name. There are a few recent language designs that provide related expressiveness, normally including separate compilation, e.g., interface and method introductions in AspectJ [28], local rebinding in Classboxes and Classbox/J [3, 2], models in F∗ [51] and views in Scala [45, 46].

4.2 Multi-class modules versus multi-parameter type classes

Various notions of multi-class modules in OO languages have been developed, ranging from lightweight constructs such as packages or namespaces to sophisticated module constructs such as virtual classes or multiple dispatch.

In abstract terms, single-parameter type classes enable the formation of sets of types with common methods, while multi-parameter type classes enable the formation of relations on types, again with common methods. Each instance injects a new relationship between specific types into the relation.

Multi-parameter type classes differ essentially from several OO generalizations of inheritance and polymorphism for multi-parameter classes with regard to the granularity of the modularization and the extensibility. That is, type-class instances can be defined for every combination of types. By contrast, mixin layers [53] or virtual classes [15, 1, 14] use some forms of nested definitions, which ties together the participants more strictly. The nesting restriction of mixin layers and virtual classes is equivalent to saying that the functional dependencies of a type class must form a tree. For example, a functional dependency a → b, c → d can be mirrored by the nesting structure (a(b(c(d))), but a → b, b → a or a b → c cannot be expressed by nesting.

Since the notion of retroactive interface extension also carries over to the multi-parameter case, type classes can also be compared to generalizations of the adapter pattern to a multi-type scenario, such as adapters in [39] or bindings in CaesarJ [1]. The differences are again similar to the comparison for the single-parameter case (see Sec. 2.2): Bindings and adapters are used explicitly, can have state and exist per combination of objects, whereas instance declarations are used implicitly, are stateless, and exist per combination of types.

4.3 Default implementations

Type classes provide a convenient way of sharing implementations among instances. That is, one can provide defaults for type-class methods in the type-class declaration. Fig. 18 illustrates this capability for the Eq class. It provides defaults for both member functions so that a type-specific implementation of either (==) or (/=) would be sufficient to implement the entire interface. In a sense, type-class declarations are therefore reminiscent of abstract classes in mainstream OO languages. We reckon that the capability of default implementations for interface methods would be a straightforward extension of the typical OO interface construct (in C# or Java). The language constructs of traits [11] provides a related extension to OO that allows to encode interfaces with default method implementations.

4.4 Explicit vs. implicit subtyping

Type-class-bounded polymorphism does not readily provide the flexibility of OO-like implicit subtyping. Fortunately, we can work around this issue. Let us consider an illustrative example; we use the adapter example from Sec. 2.2. Suppose we want to write a function that takes a list of shapes and computes a list of their circumferences. Here is an attempt:

```haskell
class Eq a where
  (==), (/=) :: a -> a -> Bool
--Minimal complete definition: (==) or (/=)
x == y = not (x /= y)
x /= y = not (x == y)
```

Figure 18. A type class with defaults

This function would work well for a list of text views, e.g.:

```haskell
test1 = allcircumference [ (TextView "" 0 0 10 10) 
test2 = allcircumference [ (TextView "" 5 5 20 20) ]
```

However, the function cannot be applied to a heterogeneous list of shapes such as a list that contains both text and image views. In fact, it is not even possible to construct such a list because the elements of a list must be of the same type. A similar problem would arise for mutable variables — without implicit subtyping, the type of the variable would be fixed by the first assignment. With implicit subtyping, a variable may reference objects of different types during its lifetime.
data AnyShape = forall a. Shape a => AnyShape a

allcircumference :: [AnyShape] -> [Int]
allcircumference = map ((AnyShape s) -> circumference s)

test2 = allcircumference [ AnyShape (TextView "0 0 10 10") , AnyShape (ImageTextView ...)]

Figure 19. Heterogeneous lists based on existential types

public boolean intersectMany(Shape[] shapes) {
    for (int i = 0; i < shapes.length; i++) {
        for (int j = i+1; j < shapes.length; j++) {
            if (shapes[i].intersect(shapes[j])) { return true; }
        }
    }
    return false;
}

Figure 20. Pairwise intersection tests for a list of shapes

We can enable these scenarios with the use of existential types [34] that make the actual type, such as a text or image view, opaque. One can say that we use explicit subtyping. In our example, we need an existential envelope for shapes; cf. the declaration of AnyShape in Fig. 19. Values that are wrapped by AnyShape can be collected in normal lists. In contrast to implicit subtyping, we need to perform an explicit up-cast by wrapping (as in AnyShape (TextView "0 0 10 10"). In the figure, we also explicitly carry out unwrapping by pattern matching (cf. the lambda expression \((AnyShape s) -> ...\)). This step does not need to be placed in 'user code' because we could add an instance for the wrapper type AnyShape to the type class Shape.

The existential wrapping approach does not work well for multiparameter type classes. As an example, let us continue the intersection example from Sec. 3.1; we want to define a function, intersectMany, that processes a collection of shapes, to see whether any pair of shapes intersect. Fig. 20 shows the (Multi)Java version. In Haskell, with an encoding that uses a single closed datatype of shapes, we have no problem whatsoever to perform the same operation (using double recursion in place of the nested loop); cf. Fig. 21.

For the type-class-based approach, we attempt this type:

\[
\text{intersectMany} :: \text{[AnyShape]} \to \text{Bool}
\]

There are two options. Option 1: suppose there is no generic default instance for \(\text{Intersect}\). In this case, the bound of AnyShape is insufficient to constrain the function intersectMany. That is, the function may encounter pairs of shapes for which no \(\text{Intersect}\) instance can be selected, which is clearly inconsistent with static typing. Option 2: suppose that the multiple dispatch scenario at hand allowed us to define a generic default instance. Hence, one may expect the bound for the type class \(\text{Shape}\) to be sufficient for applying intersectMany to any pair of shapes. That is, the type system could rest on the assumption that the default instance can be applied as ‘a last resort’. Some Haskell implementations can be actually persuaded to accept the above type subject to appropriate compiler switches.\(^3\) However, the default instance would always be applied since the opaque types cannot possibly participate in any instance selection (because they are \(\text{opaque}\)). Clearly, the universal selection of the default instance defeats the entire purpose of multiple dispatch.

There is an entirely different approach that can be put to work; it is inspired by the HList library for heterogeneous collections in


\[
\text{intersectMany} :: \text{[Shape]} \to \text{Bool}
\]

\[
\text{intersectMany} \[] = \text{False}
\]

\[
\text{intersectMany} \langle x | [] \rangle = \text{False}
\]

\[
\text{intersectMany} \langle x, y | z \rangle =
\]

\[
\text{intersect } x y \mid \text{intersectMany } (x, z) \mid \text{intersectMany } (y, z)
\]

Figure 21. Haskell variation on Fig. 20 w/o type classes

\[
\text{class } \text{IntersectMany } x \text{ where}
\]

\[
\text{intersectMany } :: x \to \text{Bool}
\]

\[
\text{instance } \text{IntersectMany } () \text{ where}
\]

\[
\text{intersectMany }_\_ = \text{False}
\]

\[
\text{instance } \text{Shape } x \Rightarrow \text{IntersectMany } (x, ()) \text{ where}
\]

\[
\text{intersectMany } x \_ = \text{False}
\]

\[
\text{instance } ( \text{Intersect } x y, \text{IntersectMany } (x, z) ),
\]

\[
\text{IntersectMany } (y, z) \Rightarrow \text{IntersectMany } (x, (y, z)) \text{ where}
\]

\[
\text{intersectMany } (x, (y, z)) =
\]

\[
\text{intersect } x y \mid \text{intersectMany } (x, z) \mid \text{intersectMany } (y, z)
\]

Figure 22. Subtyping based on heterogeneous lists

\[
\text{class } \text{Eq } a \text{ where}
\]

\[
\text{==} :: a \to a \to \text{Bool}
\]

\[
\text{class } \text{Read } a \text{ where}
\]

\[
\text{read} :: \text{String} \to a
\]

\[
\text{class } \text{Show } a \text{ where}
\]

\[
\text{show} :: a \to \text{String}
\]

Figure 23. Different uses of type-class parameters

Haskell [32] — a library that heavily relies on type-class-based programming. Essentially, we need to keep track of the precise type of all shapes when we assemble heterogeneous lists. This can be achieved by representing lists of shapes as nested, right-associative, explicitly terminated pairs. The operation intersectMany must then be defined as a type-class-based function performing induction on the nested pairs. For instance, a list with two elements would be represented as a nested tuple like this:

\[
(\text{Rectangle 1 1 2 2 }, (\text{Circle 1 1 2 }, ()))
\]

We use () to explicitly terminate the nested pairs. Fig. 22 shows the type-class-based operation intersectMany. The earlier equations of the closed-world version are lifted to the type-class level very directly. Essentially, the type-class-based version recurses into the nested tuple (at the type level) in the same way as the closed-world version recurses into the normal list at the value level.

4.5 Explicit references to MyType

The formal type-class parameter can be (in fact: must be) explicitly referenced in the member signatures; it can occur both in argument and result positions; it can also occur several times; cf. Fig. 23. Interfaces of mainstream languages like Java and C# do not admit explicit reference to the implementing type; this type is implicitly assumed for the receiver of the instance methods of the interface. More elaborate type systems and languages such as PolytOIL [5] have been proposed to admit explicit reference to ‘MyType’. Multiple references (in fact, double references) are needed for binary methods such as (\(==\)).

The use of a type-class parameter in the result position of a method requires special attention. The absence of an argument whose type can drive instance selection implies that the result type of the method application must be constrained sufficiently, e.g., by an explicit type annotation. For instance, when using read, we must disambiguate the result type of read.

```
haskell > read "True" :: Bool
True
```
5. Conclusion

We have shown that type classes enable interesting and new solutions to various software extension and integration problems. This insight is worth communicating because most of the discussed problems have been studied predominantly on the grounds of OO languages, and they have triggered a considerably amount of OO language extensions. While our analysis suggests that type classes provide a principled mechanism for software extension and integration, it also pinpoints several limitations of type classes in this context. We hope that our work promotes future research on the notion of type classes and contributes to a better understanding of the relation between advanced OO and functional (type-class-based) programming with particular focus on principled challenges for software extension and integration.

Acknowledgments

We thank the GPCE 2006 referees for their fair and insightful comments and requests. We acknowledge discussions with Vesa Karvonen and Oleg Kiselyov.

References


