# 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.

*GPCE'06* October 22–26, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-237-2/06/0010...$5.00

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, ...):

```
data Nat = Zero | Succ Nat
```

The type Nat becomes a member of the type class Eq as follows:

```
instance Eq Nat where -- Structural equality
  Zero     == Zero     = True
  Zero     == (Succ _) = False
  (Succ _) == Zero     = False
  (Succ x) == (Succ y) = x == y
```

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:

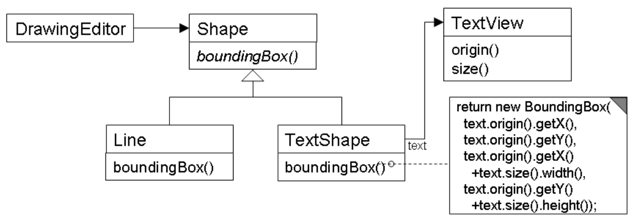
```
circumference (TextView "foo" 0 0 10 10)
```

***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 adaptee type TextView is equipped with a Shape implementation by means of the instance declaration. Wrapping is known to cause object schizophrenia due to confusion between the identity of the wrapper and the wrappee. Also, when an adaptee 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



```
return new BoundingBox(
  text.origin().getX(),
  text.origin().getY(),
  text.origin().getX()
    +text.size().width(),
  text.origin().getY()
    +text.size().height());
```

**Figure 1.** Adapter example from [16]

```
-- A datatype for the TextView class
data TextView = TextView String Int Int Int Int
origin (TextView _ x y _ _) = (x,y)
size   (TextView _ _ _ x y) = (x,y)

-- A datatype for the Point class
data Point = Point Int Int

-- A type class for the OO interface of shapes
class Shape s where boundingBox :: s -> (Point, Point)

-- The TextShape adapter
instance Shape TextView where
  boundingBox t = (Point x1 y1, Point (x1+x2) (y1+y2))
    where (x1,y1) = origin t; (x2,y2) = size t

-- A polymorphic function on shapes
circumference :: Shape s => s -> Int
circumference s = 2 * ((x2-x1) + (y2-y1))
 where (Point x1 y1, Point x2 y2) = boundingBox s
```

**Figure 2.** Type-class encoding of the adapter example

```
data ImageTextView = ImageTextView Image TextView
data Image         = ... -- details elided
leftupper :: Image -> (Int, Int)

instance Shape ImageTextView where
  boundingBox (ImageTextView im tv) = (Point x1 y1, p2)
    where (x1,y1) = leftupper im
          (p1,p2) = boundingBox tv
```

**Figure 3.** An aggregated adaptee — The datatype ImageTextView has a text view as one of its components and is itself adapted to the Shape type class. In order to calculate the bounding box of an image text view, we would like to use the bounding box of the aggregated text view. In the OO version, we cannot directly invoke boundingBox on the aggregated object; we end up creating a transient TextShape object that wraps the TextView object.

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

```
-- View of the lumberjack
class Tree t where assessedValue :: t -> Integer
class Tree t => HardWood t
class Tree t => SoftWood t

-- View of the bird
class Plant p where foodValue :: p -> Integer
class Plant p => NectarPlant p where sweetness :: p -> Integer
class Plant p => InsectPlant p
```

**Figure 4.** Separate classification hierarchies

```
-- A cherry tree has a size
data Cherry = Cherry Integer

-- Mapping to lumberjack's view
instance Tree Cherry where assessedValue (Cherry s) = s * 42
instance HardWood Cherry

-- Mapping to bird's view
instance Plant Cherry where foodValue (Cherry s) = s * 23
instance NectarPlant Cherry where sweetness (Cherry s) = 99
```

**Figure 5.** Classifying a cherry tree

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 surprisingly convenient, when compared to the approaches that were designed to address the problem of the tyranny of the dominant decomposition, i.e., subject-oriented programming or hyperspaces [20, 47, 56]. That is, these approaches require an explicit merger of the relevant parts of the original system that is supposed to incorporate the new perspective. With type classes, there is no distinction between two versions of the code that coexist (the version without the additional perspective vs. the new version). The type-class instances for a new perspective can be imported by any module and are henceforth part of the 'global instance pool'.

However, type classes do not cover the full spectrum of expressiveness in subject-oriented programming or hyperspaces. In particular, type classes are not readily useful in merging the behaviors of different perspectives. With Hyper/J, one can use different rules such as 'merge by name', which means that all methods with the same name in two perspectives are combined such that both method bodies are always executed if one of the methods is called.

## 2.4 The expression problem

The 'expression problem' [58] is the problem of achieving extensibility in both of the dimensions *data* and *operations*. In the case of OO, extensibility in the data dimension is straightforward due to the mechanism of class inheritance; extensibility in the operations dimension (read as 'new methods') is not admitted by the basic notion of closed classes. Conversely, functional programming readily admits the introduction of new operations on existing datatypes, while the introduction of new data variants is not admitted by the basic notion of closed algebraic datatypes. The expression problem and related language expressiveness have been studied in depth [58, 60, 57, 36].

```
-- A closed datatype for expression forms
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) = putStr (show i)
print (Add l r) = do print l; putStr " + "; print r
```

**Figure 6.** Extensibility in the operation dimension

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

-- 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

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 dimentions 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.

```
-- Another expression form
data     Exp x => Neg x = Neg x
instance Exp x => Exp (Neg x)

-- The extended evaluation operation
instance Eval x => Eval (Neg x) where
  eval (Neg x) = - (eval x)

-- Another operation - for printing
class Exp x => Print x where
  print :: x -> IO ()
instance Print Lit where
  print (Lit i) = putStr (show i)
instance (Print l, Print r) => Print (Add l r) where
  print (Add l r) = do print l; putStr " + "; print r
instance Print x => Print (Neg x) where
  print (Neg x) = do putStr "(- "; print x; putStr ")"
```

**Figure 8.** Extensions in the data and operation dimensions

Fig. 8. Combination of extensions boils down to module import. The approach is statically safe in so far that one cannot possibly invoke operations on variants that are not covered. The incomplete implementation of an operation is revealed (statically) by any attempt to apply the operation to an uncovered variant. This laziness may be arguable; one may favor a guarantee of completeness, even without 'user code' that exercises an operation per case.

A weakness of the approach is that term structure is fully reified at the type level; this challenges the convenience of inferred types and the performance of classic instance selection; cf. [33]. Also, values of a given extensible datatype cannot be aggregated or stored in a straightforward manner. Furthermore, the definition of functions that *return* values of the extensible datatype cannot adopt the aforementioned recipe. We reconsider several of these restrictions in Sec. 4.

A fundamental disadvantage of the type-class-based solution is that extensibility has to be enabled by adherence to an anticipating style, when compared to designated language constructs for extensible (open) datatypes and functions [44, 35].

## 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. We may want to implement these different combinations as modular contributions to the following (Java) method:

```
-- A closed datatype of shapes
data Shape = Rectangle Int Int Int Int | Circle Int Int Int

-- A closed function for intersection
intersect :: Shape -> Shape -> Bool
intersect (Rectangle x1 x2 y1 y2) (Rectangle a1 a2 b1 b2) = ...
intersect (Circle x y r) (Circle x2 y2 r2) = ...
intersect (Circle x y r) (Rectangle x1 x2 y1 y2) = ...
intersect r@(Rectangle _ _ _ _) c@(Circle _ _ _) =
  intersect c r -- symmetric handling of CxR and RxC
```

**Figure 9.** Non-extensible, value-level dispatch

```
-- Datatypes for different kinds of shapes
data Rectangle = Rectangle Int Int Int Int
data Circle = Circle Int Int Int

-- A type class that comprehends all shapes
class Shape x
instance Shape Rectangle
instance Shape Circle

-- A type class for a multiple dispatch
class (Shape x, Shape y) => Intersect x y where
  intersect :: x -> y -> Bool

-- A special case for RxR ( others omitted for brevity )
instance Intersect Rectangle Rectangle where
  intersect r r' = ...
```

**Figure 10.** Extensible, type-level dispatch

```
public class Shape {
  ...
  public boolean intersect (Shape s) { ... }
}
```

There are common approaches to encode multiple dispatch while only using single dispatch: (i) case discrimination is performed on the dynamic type of arguments using instanceof checks; (ii) multiple dispatch is scattered over multiple messages and auxiliary objects as in the visitor pattern. Both approaches have well-known disadvantages, in particular extensibility problems; see [8, 10] for a discussion.

Multiple dispatch (as in MultiJava) supports the modular definition of methods by allowing the modular contribution of cases that are specific to receiver and argument types. The idea is that *the most specific method implementation is chosen at runtime*. For instance, the special case for the intersection of two rectangles is defined as follows (in MultiJava); notice the special syntax Shape@Rectangle, which restricts the runtime argument type to a Rectangle:

```
public class Rectangle extends Shape {
  ...
  public boolean intersect (Shape@Rectangle r) {
    /* efficient code for two Rectangles */
  }
}
```

Fig. 9 shows a trivial Haskell model of the intersection sample *while assuming closed datatypes*. There is one equation for each combination of shapes. Basic pattern matching facilitates (multiple) dispatch at the value level; there is an apparent code modularization (with the modules being the equations), but extensibility is absent: the datatype Shape and the function intersect are closed.

Fig. 10 shows a type-class-based solution which essentially lifts multiple dispatch to the type level. Each kind of shape is modeled 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:

```
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:

```
instance Shape x => Intersect Rectangle x where ... -- RxS
instance Shape x => Intersect x Rectangle 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 SxR instance
intersect r c -- selects RxS instance
intersect r r -- static error; ambigious 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 RxR 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],

```
class Eq v => Graph g v e | g -> v e where
  vertices :: g -> e -> (v,v)
  edges    :: g -> 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

```
instance Graph [[Int]] Int (Int,Int) where
  vertices g (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 _ v1 v2) = (v1,v2)
  edges g (Vertex _ es ) = es
```

**Figure 12.** Two different graph families

```
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 type class Graph, one using integer lists, and one using algebraic datatypes.[2] 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:

[2] The **deriving** Eq annotation 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.

```
abstract class Graph {
  abstract class Edge {
    Vertex getLeft ();
    Vertex getRight ();
  }
  abstract class Vertex { ... }
}
class GraphAlg {
  g.Vertex otherVertex( final Graph g, g.Vertex v, g.Edge e) { ... }
}
class ConcreteGraph1 extends Graph {
  class Edge {
    private Vertex v1;
    private Vertex v2;
    Edge(Vertex v1, Vertex v2) {
      this.v1 = v1; this.v2 = v2;
    }
    Vertex getLeft() { return v1; }
    Vertex getRight() { return v2; }
  }
  class Vertex { ... }
}
class ConcreteGraph2 extends Graph { ... }

 // Some test code
 final Graph g1 = new ConcreteGraph1();
 final Graph g2 = new ConcreteGraph2();
 g1.Vertex v1 = new g1.Vertex(...);
 g2.Vertex v2 = new g2.Vertex(...);
 g1.Vertex e1 = new g1.Edge(...);
 g2.Vertex e2 = new g2.Edge(...);

 GraphAlg alg = new GraphAlg();

 g1.Vertex test1 = alg.otherVertex(g1,v1,e1);
 g2.Vertex test2 = alg.otherVertex(g2,v2,e2);
```

**Figure 14.** The graph example rephrased in CaesarJ [41, 1] — Is is important to notice that all nested classes in CaesarJ are virtual classes and hence class-valued properties of the enclosing object.

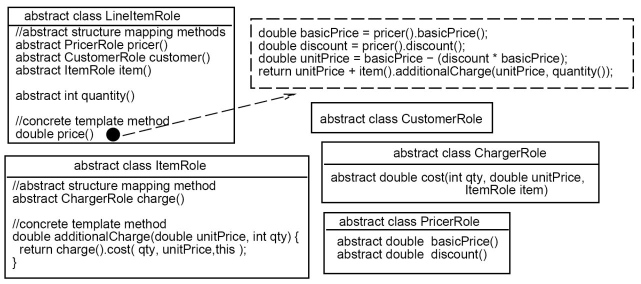
```
Graph g = complexTest() ? new ConcreteGraph1()
                        : new ConcreteGraph2();
alg.otherVertex(g, new g.Vertex(), new g.Edge());
```

Hence, virtual classes are more flexible than our type-class-based transcription, if we assume that values can be passed and computed and stored more flexibly than types, which holds for most programming languages. One may engage in *type-level programming* using type classes [19, 38, 30, 32, 31], even though this step may imply considerable encoding efforts.

## 3.3 Framework integration

The integration of OO frameworks into an existing application or the combination of multiple frameworks are well-known problems [37, 43, 21, 42, 4]. When frameworks are specialized by subclassing framework classes, the integration of a framework into an existing application requires elaborate adapters. Not even multiple inheritance (i.e., creating a subclass of the framework class that is also a subclass of the application class) is sufficient in this case because (apart from the typical multiple inheritance problems) the existing application contains hard links to the original application class in superclass declarations or constructor invocations.

Fig. 15 shows an example of an OO framework for pricing. Fig. 16 shows a type-class-based encoding of the framework. There is functional dependency that states that the first type-class parameter, p, determines all other type parameters of the type class. Technically, the actual type for the formal parameter p is not expected to represent any interesting value domain; it merely serves a type-level value that is used to unambiguously select a type-class instance. Coexisting framework instances may use the same types for some (or all) of the framework parameters.



```
abstract class LineItemRole
//abstract structure mapping methods
abstract PricerRole pricer()
abstract CustomerRole customer()
abstract ItemRole item()

abstract int quantity()

//concrete template method
double price()
```

```
double basicPrice = pricer().basicPrice();
double discount = pricer().discount();
double unitPrice = basicPrice − (discount * basicPrice);
return unitPrice + item().additionalCharge(unitPrice, quantity());
```

```
abstract class CustomerRole
```

```
abstract class ItemRole
//abstract structure mapping method
abstract ChargerRole charge()

//concrete template method
double additionalCharge(double unitPrice, int qty) {
  return charge().cost( qty, unitPrice,this );
}
```

```
abstract class ChargerRole
abstract double cost(int qty, double unitPrice,
                     ItemRole item)
```

```
abstract class PricerRole
abstract double basicPrice()
abstract double discount()
```

**Figure 15.** Framework example from [21, 42]

```
class Pricing p lineItem pricer charger item customer
      | p −> lineItem pricer charger item customer
  where
    pricer     :: p −> lineItem −> pricer
    customer   :: p −> lineItem −> customer
    item       :: p −> lineItem −> item
    quantity   :: p −> lineItem −> Int
    basicprice :: p −> pricer −> Double
    discount   :: p −> pricer −> Double
    cost       :: p −> charger −> Int −> Double −> item −> Double
    charge     :: p −> item −> charger

additionalCharge :: Pricing p lineItem pricer charger item customer =>
                    p −> item −> Double −> Int −> Double
additionalCharge p item unitPrice qty =
  cost p (charge p item) qty unitPrice item

price :: Pricing p lineItem pricer charger item customer =>
         p −> lineItem −> Double
price p l = unitPrice +
          (additionalCharge p (item p l) unitPrice (quantity p l))
  where
    basicPrice = basicprice p ( pricer p l)
    dcount = discount p ( pricer p l)
    unitPrice = basicPrice − dcount * basicPrice
```

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

```
data RegularPricing = RP

instance Pricing RegularPricing
            Quote (Quote,HWProduct) [Tax] HWProduct Customer
  where ... −− omitted for brevity
```

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

Fig. 17 alludes to a binding of the framework to a particular set of existing data structures. The trivial datatype RegularPricing is used to select the particular framework instance. The other parameters of the Pricing type class are represented by datatypes that are supposed to play the particular roles defined in the type class. For instance, Quote plays the role of lineItem. A framework function can now be called with values of adapted application types, whereby the first parameter selects the framework binding, e.g., price RP myQuote (myQuote,myHWPr) [] myHWPr cust.

***Assessment*** This kind of mapping is non-invasive and very flexible because neither the framework nor the application need to be modified for the integration. The mapping is not restricted to a nominal 1:1 mapping; instead structural types may be associated with the type parameters as well. For instance, in Fig. 17, the charger role is played by *lists* of taxes, i.e., [Tax]. In other respects, the OO approach and the type-class approach are very similar: Instead of overriding abstract methods of a framework class in an application-specific subclass, we define the methods of a type class in the application-specific type-class instance.

It has been observed [17] that plain multi-parameter type classes with functional dependencies can become tedious when used for the encoding of type associations. The main problem is that type parameters for associated types have to be explicitly enumerated whenever the type class is referenced in a constraint — even when a particular function does not interact with all the type parameters. (The signature of price in Fig. 16 alludes to this problem.) Accordingly, enhancements of Haskell's type classes have been proposed: so-called associated types [7] and (synonyms [6]).

## 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 $F^G$ [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 multiple 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, c −> d can be mirrored by the nesting structure a(bc(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*.

```
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

### 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:

```
allcircumference  :: Shape s => [s] −> [Int]
allcircumference  =  map circumference
```

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

```
test1 = allcircumference [ (TextView "" 0 0 10 10)
                         , (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 multi-parameter 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:

```
intersectMany :: [AnyShape] -> Bool
intersectMany = ...
```

There are two options. Option 1: suppose there is no generic default instance for 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 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 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 *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

[3] See the related bug report for GHC: `http://www.haskell.org//pipermail/glasgow-haskell-bugs/2006-July/006809.html`.

```
intersectMany :: [Shape] -> Bool
intersectMany []      = False
intersectMany (x:[])  = False
intersectMany (x:y:z) =
  intersect x y || intersectMany (x:z) || intersectMany (y:z)
```

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

```
class IntersectMany x where
  intersectMany :: x -> Bool
instance IntersectMany () where
  intersectMany _ = False
instance Shape x => IntersectMany (x,()) where
  intersectMany _ = False
instance ( Intersect x y, IntersectMany (x,z),
        IntersectMany (y,z) ) => IntersectMany (x,(y,z)) where
  intersectMany (x,(y,z)) =
    intersect x y || intersectMany (x,z) || intersectMany (y,z)
```

**Figure 22.** Subtyping based on heterogeneous lists

```
class Eq a    where  (==) ::  a -> a -> Bool
class Read a  where  read ::  String -> a
class Show a  where  show ::  a -> 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:

```
( Rectangle 1 1 2 2, ( 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-prompt> read "True" :: Bool
True
```

```
-- Trees for untyped representation
data Tree = Node String [Tree]

-- Parsing of trees
class FromTree x where
  fromTree :: Tree -> x
instance FromTree Int where
  fromTree (Node s []) = read s
instance FromTree Lit where
  fromTree (Node "Lit" [i]) = Lit (fromTree i)
instance (Exp e, Exp e', FromTree e, FromTree e')
        => FromTree (Add e e') where
  fromTree (Node "Add" [x,y]) = Add (fromTree x) (fromTree y)
instance (Exp e, FromTree e) => FromTree (Neg e) where
  fromTree (Node "Neg" [x]) = Neg (fromTree x)
```

**Figure 24.** A failed attempt at open parsing

```
onTree :: ( forall x. Exp x => x -> y) -> Tree -> y
onTree f (Node "Lit" [i])    = f (Lit (fromTree i))
onTree f (Node "Neg" [x])    = onTree (f . Neg) x
onTree f (Node "Add" [x,y])  = onTree (\x' ->
                                 onTree (f . Add x') y) x
```

**Figure 25.** Parsing based on CPS — The case discrimination on trees leads to the construction of the appropriate expression form, which is not returned however, but instead processed by the provided argument function. This function must be necessarily polymorphic so that it can be applied to every possible expression. This implies a rank-2 polymorphic type for onTree, as shown.

When we adopt this idea to the extensible datatypes of Sec. 2.4, then we encounter a challenge. Consider the problem of 'parsing' (or de-serializing) expressions, while using 'untyped' trees as input. If we follow the same recipe as for the evaluation and print functions in Fig. 7 and 8, we end up with the code in Fig. 24. However, the shown definition is practically useless because we would need to annotate the parse expression with the precise expression type whose shapes coincides with the shape of the actual expression. For instance, the following annotation is impractical, and leaving it out would cause an ambiguous expression:

```
-- How to know that we are expecting a literal?
myExp = (fromTree sometree) :: Lit
```

To summarize, we cannot scatter the cases for printing over multiple instances because there is no useful way of driving instance selection. However, we can provide a closed parse function with a single type. In fact, there are two equivalent approaches: (i) wrap the parsing result in an existential envelope; (ii) use continuation-passing style (CPS) to process the parsing result; cf. Fig. 25 for an illustration of the latter approach. Both approaches suffer from two weaknesses: (i) unless we engage in encoding such that we defer taking the fixpoint of the recursive parse function, the function is nonextensible; (ii) the parse function is also closed over the bound on the parsing result (such as Exp or Print).

Both limitations call for future work; we only record some hypotheses here. Regarding (i): we would like to avoid the hardwired enumeration of Lit, Add and Neg in Fig. 25. To this end, we would need either strong forms of reflection or means of modular registration of variants. Both options are known in the object world; think of serialization frameworks, for example. Regarding (ii): we would like to be able to use the parse functions on new (open) functions without anticipating their bounds. To this end, we would either need a form of type-class parameterization [22] or a form of 'dependent type classes' where we can express that a new class (such as Print) is not just a subclass of an existing class (such as Exp), but the inverse instance-relationship holds, too.

## 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

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *LNCS*, pages 135–173. Springer, 2006.

[2] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/j: controlling the scope of change in java. In R. Johnson and R. P. Gabriel, editors, *OOPSLA*, pages 177–189. ACM Press, 2005.

[3] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A Minimal Module Model Supporting Local Rebinding. In L. Böszörményi and P. Schojer, editors, *Modular Programming Languages, Joint Modular Languages Conference, JMLC 2003*, volume 2789 of *LNCS*, pages 122–131. Springer, 2003.

[4] L. M. Berlin. When objects collide: Experiences with reusing multiple class hierarchies. In *ECOOP/OOPSLA'90; ACM SIGPLAN Notices 25(10)*, pages 181–193, 1990.

[5] K. Bruce, L. Cardelli, G. Castagna, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[6] M. M. T. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In *ICFP '05*, pages 241–253. ACM Press, 2005.

[7] M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *POPL'05*, pages 1–13. ACM Press, 2005.

[8] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP'92*, volume 615 of *LNCS*, pages 33–56. Springer, 1992.

[9] K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *ACM Conference on LISP and Functional Programming*, pages 170–181. ACM Press, 1992.

[10] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *OOSPLA'00; ACM SIGPLAN Notices 35(10)*, pages 130–145. ACM Press, 2000.

[11] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2):331–388, Mar. 2006.

[12] G. Duck, S. Peyton Jones, P. Stuckey, and M. Sulzmann. Sound and Decidable Type Inference for Functional Dependencies. In D. Schmidt, editor, *ESOP'04*, volume 2986 of *LNCS*, pages 49–63. Springer, 2004.

[13] E. Ernst. Family Polymorphism. In *ECOOP'01*, volume 2072 of *LNCS*, pages 303–326. Springer, 2001.

[14] E. Ernst. Higher-order hierarchies. In *Proceedings ECOOP '03*, LNCS. Springer, 2003.

[15] E. Ernst, K. Ostermann, and W. Cook. A Virtual Class Calculus. In *POPL'06*, pages 270–282. ACM Press, 2006.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[17] R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. In *OOPSLA'03*, pages 115–134. ACM Press, 2003.

[18] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. Wadler. Type Classes in Haskell. *ACM TOPLAS*, 18(2):109–138, 1996.

[19] T. Hallgren. Fun with functional dependencies. In *Joint Winter Meeting of the Departments of Science and Computer Engineering, Chalmers University of Technology and Goteborg University, Varberg, Sweden, Jan. 2001*, 2001. `http://www.cs.chalmers.se/~hallgren/Papers/wm01.html`.

[20] W. Harrison and H. Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *OOPSLA'93; ACM SIGPLAN Notices 28(10)*, pages 411–428. ACM Press, 1993.

[21] I. Holland. *The Design and Representation of Object-Oriented Components*. PhD thesis, Northeastern University, Department of CS, 1993.

[22] J. Hughes. Restricted data types in Haskell. In E. Meijer, editor, *Haskell Workshop*, 1999.

[23] U. Hölzle. Integrating independently-developed components in object-oriented languages. In *ECOOP'93*, volume 707 of *LNCS*, pages 36–56. Springer, 1993.

[24] M. Jones. A theory of qualified types. In B. Krieg-Brückner, editor, *ESOP'92*, volume 582 of *LNCS*, pages 287–306. Springer, 1992.

[25] M. Jones. Type Classes with Functional Dependencies. In *ESOP'00*, pages 230–244. Springer, 2000.

[26] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA'93*, pages 52–61. ACM Press, 1993.

[27] S. Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *ESOP'88*, volume 300 of *LNCS*, pages 131–144. Springer, 1988.

[28] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP'01*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[29] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[30] O. Kiselyov and C. chieh Shan. Functional pearl: implicit configurations–or, type classes reflect the values of types. In *ACM SIGPLAN Workshop on Haskell*, pages 33–44. ACM Press, 2004.

[31] O. Kiselyov and R. Lämmel. Haskell's overlooked object system. Draft; submitted for journal publication; available online at `http://www.cwi.nl/~ralf/OOHaskell/`, 2005.

[32] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *ACM SIGPLAN Workshop on Haskell*, pages 96–107. ACM Press, 2004. See `http://www.cwi.nl/~ralf/HList/` for an extended technical report and for the source distribution.

[33] R. Lämmel. Re: Extensible grammars, 2004. Post to `comp.compilers`; `http://compilers.iecc.com/comparch/article/04-12-111`; Follow-up on `haskell.org`; `http://www.haskell.org/pipermail/haskell/2005-February/015346.html`.

[34] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM TOPLAS*, 16(5):1411–1430, 1994.

[35] A. Löh and R. Hinze. Open data types and open functions. In *PPDP'06*, pages 133–144. ACM Press, 2006.

[36] R. E. Lopez-Herrejon, D. S. Batory, and W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In A. P. Black, editor, *ECOOP*, volume 3586 of *LNCS*, pages 169–194. Springer, 2005.

[37] M. Mattson, J. Bosch, and M. E. Fayad. Framework integration problems, causes, solutions. *Communications of the ACM*, 42(10):80–87, October 1999.

[38] C. McBride. Faking It (Simulating Dependent Types in Haskell). *Journal of Functional Programming*, 12(4–5):375–392, 2002.

[39] M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *OOPSLA'98; ACM SIGPLAN Notices 33(10)*, pages 97–116. ACM Press, 1998.

[40] M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings OOPSLA '02, ACM SIGPLAN Notices 37(11)*, pages 52–67, 2002.

[41] M. Mezini and K. Ostermann. Conquering Aspects With Caesar. In *AOSD'03*, pages 90–99. ACM Press, 2003.

[42] M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters. In M. Aksit, editor, *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer, 2001.

[43] H. Mili, M. Fayad, D. Brugali, D. S. Hamu, and D. Dori. Enterprise frameworks: issues and research directions. *Software: Practice & Experience*, 32(8):801–831, 2002.

[44] T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. In *ICFP'02*, pages 110–122. ACM Press, 2002.

[45] M. Odersky et al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[46] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA'05*, pages 41–57. ACM Press, 2005.

[47] H. Ossher and W. Harrison. Combination of Inheritance Hierarchies. In *OOPSLA'92; ACM SIGPLAN Notices 27(10)*, pages 25–40. ACM Press, 1992.

[48] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[49] S. Peyton-Jones. *Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language*. Available from http://haskell.org, 1999.

[50] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: exploring the design space. In J. Launchbury, editor, *Haskell Workshop*, 1997.

[51] J. G. Siek and A. Lumsdaine. Essential language support for generic programming. In *PLDI'05*, pages 73–84. ACM Press, 2005.

[52] J. G. Siek and A. Lumsdaine. Language Requirements for Large-Scale Generic Libraries. In R. Glück and M. R. Lowry, editors, *GPCE'05*, volume 3676 of *LNCS*, pages 405–421. Springer, 2005.

[53] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin-layers. In *ECOOP'98*, volume 1445 of *LNCS*, pages 550–570. Springer, 1998.

[54] C. Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings 19th Australian Computer Science Conference*. Australian Computer Science Communications, 1996.

[55] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[56] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE'99*, pages 107–119. ACM Press, 1999.

[57] M. Torgersen. The Expression Problem Revisited. In *ESOP'04*, volume 3086 of *LNCS*, pages 123–143. Springer, 2004.

[58] P. Wadler. The expression problem. Message to java-genericity electronic mailing list, November 1998. Available online at `http://www.daimi.au.dk/~madst/tool/papers/expression.txt`.

[59] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL'89*, pages 60–76. ACM Press, 1989.

[60] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. Technical Report IC/2004/33, École Polytechnique Fédérale de Lausanne, 2004.