Objects to Unify Type Classes and GADTs

Bruno C. d. S. Oliveira
Oxford University Computing Laboratory
Wolfson Building, Parks Road,
Oxford OX1 3QD, UK
{bruno}@comlab.ox.ac.uk

Martin Sulzmann
IT University of Copenhagen
Rued Langgaards Vej 7,
2300 Copenhagen, Denmark
{martin.sulzmann}@gmail.com

Abstract

We propose an Haskell-like language with the goal of unifying type classes and *generalized algebraic datatypes* (GADTs) into a single class construct. We treat classes as *first-class types* and we use *objects* (instead of type class instances and data constructors) to define the values of those classes. We recover the ability to define functions by pattern matching by using *sealed classes*. The resulting language is simple and intuitive and it can be used to define, with similar convenience, the same programs that we would define in Haskell. Furthermore, unlike Haskell, dictionaries (or objects) can be explicitly (as well as implicitly) passed to functions and we can program in a simple object-oriented style directly.

1. Introduction

Datatypes and *type classes* (Hall et al. 1996) are two of the most prominent features of Haskell. With datatypes we can define functions by *pattern matching*. That is, we perform a case analysis over the set (pattern) of datatype values. With type classes we can define type-overloaded methods and then define instances that implement the class interface. Datatypes and type classes are generally considered orthogonal concepts. The set of values belonging to a datatype is fixed but we can define an arbitrary number of functions which operate on a datatype. We say that datatypes are closed but the set of functions that can operate on them is open. On the other hand, type classes are open but the set of methods belonging to a class is closed. Openness of a type classes means that we can define new (type class) instances. In a sense, type classes provide us with an object-oriented programming style. However, support for this style is limited, since type classes were designed as an *ad-hoc polymorphism* (Strachey 1967) mechanism and not as a mechanism to support object-oriented programming.

In this paper we propose a Haskell-like language with a generalized *class* construct that retains the ability to be used as an ad-hoc polymorphism mechanism, but it can also be conveniently used to program in a simple object-oriented programming style. In our proposal, classes are first-class types and we use *objects*, instead of type class instances and data constructors, to create the values of classes. Objects come in two forms: *implicit objects* play a similar role to type class instances in Haskell and can be implicitly used and inferred in definitions; while (normal) objects are explicitly constructed and passed to definitions. The implementation of objects is basically just a slight generalization of the dictionary-passing translation (Hall et al. 1996).

We also introduce the notion of *sealed classes*, for which a closed world perspective is assumed. That is, all objects that construct values of a certain class are defined in the same module of that class and we can define new functions using pattern matching on the objects of that class. Sealed classes avoid the need for a separate *data* mechanism, while providing a unified and uniform syntax that is shared with (open) classes. Furthermore, sealed classes are more general than traditional datatypes and GADTs (Peyton Jones et al. 2006) because they can have associated methods.

In essence, classes in our proposal provide a unifying model for type classes and GADTs based on the familiar concept of classes and objects from object-oriented languages. We can still write programs with classes and objects like we would, in Haskell, with type classes and GADTs, but we can also write object-oriented style programs in an easy and convenient way. Moreover, unlike Haskell, our objects (which play the role of dictionaries) can be explicitly and implicitly passed to functions, while in Haskell dictionaries are always implicit. To summarize, the main contributions of this paper are:

- *a class system* that generalizes Haskell's type class system, allows the construction of implicit as well as explicit "dictionaries" using objects and can be used to program in a simple object-oriented style;
- *sealed classes*, which allow us to introduce new functions defined by pattern matching and preclude the need for a separate *data* construct;
- numerous examples supporting the usefulness of our system.

Overview In Section 2, we introduce the key ideas of our system. Section 3 shows some example applications that demonstrate how the extra generality of our class system allows us to easily write solutions for problems that would have somehow roundabout solutions in Haskell. In Section 4, we present a core calculus for our system. In Section 5 we discuss the relationship between what we propose and existing work and, in Section 6, we conclude and present possible future work.

2. Key Ideas

We illustrate the key ideas of our work via a few simple examples.

2.1 Classes as Types

Suppose that we want to define an abstract interface for sets supporting multiple implementations. A natural object-oriented style solution for this problem is presented in Figure 1 using our generalized notion of classes. The Set class is almost a valid Haskell class, except that we use Set as a type in some of the method signatures, which is forbidden in Haskell (causing a type-error). However in our system, classes are just types and there is no problem of using Set on type positions.

2.2 Objects as Values

In Figure 2 we show a simple (but not necessarily efficient) implementation of the set interface. We introduce value constructors to define set values by using **object** declarations. An **object** decla-

```
class Set a where
  member :: a → Bool
  merge   :: Set a → Set a
  extract :: Maybe (a, Set a)
  insert  :: a → Set a
```

Figure 1: An abstract interface for sets.

```
object ListSet :: Eq a ⇒ [a] → Set a
object ListSet l where
  member x = elem x l
  merge s   = mergeList l s
  extract   = case l of
                  []       → Nothing
                  (x : xs) → Just (x, ListSet xs)
  insert x  = ListSet (union [x] l)
mergeList :: [a] → Set a → Set a
mergeList [] s       = s
mergeList (x : xs) s = mergeList xs (insert s x)
```

Figure 2: A simple set implementation.

ration like *ListSet* is divided in two parts: a signature; and a definition. The signature (the first **object** line in the example) tells us the type of the associated value constructor for the object. In the definition we provide all the arguments required by the constructor of the object (in this case just the argument l) and, after the **where** clause, we give the definitions of the methods in the class (which is determined by the return type of the constructor). We use the value constructor *ListSet* in the definitions of *extract* and *insert* to create a *new object instance* for the *ListSet* implementation. If we want to have different implementations of sets we can just add new object declarations.

We can define functions in the same way as in Haskell. For example, we could define a function that concatenates sets as follows:

```
concatSet :: Eq a ⇒ Set (Set a) → Set a
concatSet s = case extract s of
  Nothing     → ListSet []
  Just (x, xs) → x.merge (concatSet xs)
```

In the definition of *concatSet* we use the *merge* method, which (like with Haskell type classes) has type *merge* :: *Set a* ⇒ *Set a* → *Set a*. However, unlike Haskell, we can explicitly pass a value for the dictionary argument (that is, the argument on the left of ⇒). In the above example, the *dot* notation *x.merge* says that we pass the dictionary *x* to function *merge*. In OO terminology, we would say that we access the method *merge* in the object *x*.

We illustrate how to create *objects* and use them in the following example:

```
ins :: Set a → a → Set a
ins s x = s.insert x

set :: Set Int
set = ins (ins (ins (ListSet []) 3) 4) 3

test :: Bool
test = set.member 3    -- test = True
```

We use *ins* to define a set of integers and we use the object constructor *ListSet* to create a new empty Set. The value *test* shows how we can test if 3 is a member of the set that we have just defined.

2.3 Haskell-Style Type Class Programming

Objects allow us to explicitly define values of classes, but with type classes the objects (or dictionaries) would be implicitly constructed

```
class Eq a where
  (≡) :: a → a → Bool
implicit object EqInt :: Eq Int
implicit object EqInt where
  (≡) = primIntEQ
object EqInt2 :: Eq Int
object EqInt2 where
  x ≡ y = primIntEQ (x ‘mod‘ y) 0
implicit object EqPair :: (Eq a, Eq b) ⇒ Eq (a, b)
implicit object EqPair where
  (a, b) ≡ (c, d) = a ≡ c ∧ b ≡ d
```

Figure 3: A simplified *Eq* type class and some objects.

for us. Our class system supports a type class programming style via **implicit object** declarations. These declarations have similar restrictions to type class instances and, consequently, there are less type signatures allowed for implicit objects than for (normal) objects. However, we can use the type system to implicitly build values of implicit objects for us. For example, the *ListSet* constructor has an *Eq a* constraint. In Figure 3 we show how to define a simplified version of the class *Eq a*. The **class** definition is written exactly in the same way as in Haskell. We use implicit objects to define implementations of that class. We provide two simple examples for integers and pairs and we also provide a second object that shows how we could write an alternative *Eq Int* implementation. While there can be only one *Eq t* implicit object on scope for some given type t, there can be multiple objects with that type and we can pass those explicitly (overriding the default implicit object) by using the dot notation. For example we could define

```
eqPair :: Eq (Int, Int)
eqPair = (EqInt, EqInt2).EqPair

test2 :: Bool
test2 = eqPair.(≡) (3, 8) (3, 4)    -- test2 = True
```

The idea here is that the explicitly built dictionary *eqPair* uses *EqInt* for comparing the first value of the pair and *EqInt2* for comparing the second value. Note that our system handles multiple constraints by using a tuple-semantics.

2.4 Sealed Classes

We can also program in the same way we would program with datatypes or GADTs by using sealed classes. Here is an example defining the standard lists:

```
sealed class List a where
object Nil   :: List a
object Cons :: a → List a → List a
```

We use a little bit of syntactic sugar on the definition of the *Nil* and *Cons* objects because they do not define (or override) any methods. This is just short for:

```
object Nil :: List a
object Nil where

object Cons :: a → List a → List a
object Cons x xs where
```

With sealed classes all the objects need to be defined in the same module of the corresponding class. This precludes the possibility of extending the class modularly with extra objects, but it also means that we know all the objects that belong to that class statically. We use this knowledge to write definitions by pattern matching like we would do in Haskell:

```
length :: List a → Int
length Nil          = 0
length (Cons x xs) = 1 + length xs
```

```
data Set a = Set{
  member  :: a → Bool,
  merge   :: Set a → Set a,
  extract :: Maybe (a, Set a),
  insert  :: a → Set a}
listSet :: Eq a → [a] → Set a
listSet d l = Set (λx → elem d x l)
                  (mergeList l)
                  (case l of
                     []       → Nothing
                     (x : xs) → Just (x, listSet d xs))
                  (λx → listSet d (union [x] l))
mergeList :: [a] → Set a → Set a
mergeList []       s = s
mergeList (x : xs) s = mergeList xs (insert s x)
```

Figure 4: *Set* translated via the dictionary translation.

2.5 Dictionary Translation of Classes and Objects

Our classes and objects can be translated in very much the same way that Haskell type classes can be translated via the *dictionary translation* Hall et al. (1996). The difference is that our dictionary translation is slightly more general because of the need to account for the fact that classes are first-class types and objects can take any kind of arguments (and not just dictionaries). In Figure 4 we can see how the set example could be translated using the dictionary translation. For convenience, we use Haskell's labelled datatypes in the translation.

Some readers may ask why not program directly in this "dictionary-passing" style. There are three main reasons for justifying the introduction of a new class system instead of just using the dictionary translation encoding directly.

- With the class system and with the dot notation that we propose we can easily pass dictionary values implicitly or explicitly. This adds important convenience and solves limitations of Haskell's class system pointed out in the past (Kahl and Scheffczyk 2001; Dijkstra and Swierstra 2005). Moreover, because in our system datatypes are just defined using sealed classes, we can easily have implicitly constructed datatype values in the style of Omega's propositions (Sheard 2005).
- We allow a simple, intuitive and direct object-oriented programming style that we can use to define, for example, ADTs like *Set*. While we could just use Haskell records directly, the fact is that the encoding does not make it clear the relationship with object-oriented programming. Something that may attest that this encoding is not intuitive is the fact that, although there are several different proposals on how to implement ADTs in the Haskell literature, we have not been able to find a single one using the object-oriented style solution we proposed.
- Finally, on the whole, we believe that the language we propose is conceptually simpler than Haskell: we do not need to have separate 'class' and 'data' constructs; type class instances and data constructors are unified into a single concept; and there is basically no distinction on which values can be passed implicitly or explicitly.

3. Applications

In this section we show some interesting applications of our system. In Section 3.1 we show how to model open datatypes. In Section 3.2 we provide a graph example of an abstract datatype and compare it to solutions for the *bulk types* (Peyton Jones 1996) problem in Haskell. Section 3.3 shows how dictionary overriding can be useful to avoid pairs of similar definitions. In Section 3.4, we discuss how we can emulate *implicit parameters* (Lewis et al. 2000). Finally, in Sections 3.5 and 3.6, we show how to model (closed) GADTs with sealed classes and demonstrate how associated methods can be useful.

```
class Format t where
  sprintf′ :: String → t
object E :: Format String
object E where
  sprintf′ = id
object I :: Format t → Format (Int → t)
object I k where
  sprintf′ s x = k.sprintf′ (s ++ show x)
object C :: Format t → Format (Char → t)
object C k where
  sprintf′ s x = k.sprintf′ (s ++ [x])
object S :: String → Format t → Format t
object S x k where
  sprintf′ s = k.sprintf′ (s ++ x)
sprintf :: Format t → t
sprintf f = f.sprintf′ ""
```

Figure 5: An open (or extensible) *Format* datatype.

3.1 Open Datatypes

The C-style *sprintf* function, which takes a variable number of parameters, has always been a challenge for programmers using strongly and statically typed languages. The problem with *sprintf* is that, in its true essence, it requires dependent types. This happens because the value of the format string determines the type of the function. However, it has been shown by Danvy (1998) that, by changing the representation of the control string, it is possible to encode *sprintf* in any language supporting a standard Hindley-Milner type system. Still, this encoding uses explicit continuation passing style. It has been suggested by Chakravarty et al. (2005b) that type functions offer a more direct, inductive definition. That solution requires one type/datatype per each possible format kind, which can be freely combined.

One criticism that can be made to both Danvy's and the associated types solution is that they are less type-safe than one would like: with Danvy's solution we are free to provide any function of the right type as a continuation; while with the associated types solution we can mix a value of some type/datatype that is not meant to be any kind of format string with other format values, which would cause an unresolved overloading error. An alternative solution is to use GADTs to encode the format string (Oliveira and Gibbons 2005), which provides a solution that does not have this type safety problem. However, unlike the other two solutions, we cannot add new format specifiers.

In Figure 5 we show how we can have an implementation of *sprintf* in our system that allows new format specifiers to be added, is type-safe and it is written in direct-style. Therefore combining the advantages of the solutions discussed above, while also being shorter and (in the authors opinion) clearer than the associated types solution. The types of the format objects have the same types of the value constructors of an equivalent (Haskell) *Format* GADT solution, which allows us to build format specifiers like:

```
format :: Format (Int → Char → String)
format = S "Int: " $ I $ S ", Char: " $ C $ S "." E
```

```
class Graph node where
  outEdges :: node → [(node, node)]

object AdjList :: Enum v ⇒ [[v]] → Graph v
object AdjList g where
  outEdges v = [(v, w) | w ← g !! fromEnum v]

type AdjMat = Array.Array (Int, Int) Bool

object AdjMatG :: AdjMat → Graph Int
object AdjMatG g where
  outEdges v = let ((from, _), (to, _)) = bounds g
               in [(v, w) | w ← [from .. to], g ! (w, v)]
```

Figure 6: An abstract type and two implementations for Graphs.

Note that, alternatively, we could have defined E, I and C as implicit objects by giving them the following types:

```
implicit object E :: Format String
implicit object I  :: Format t ⇒ Format (Int → t)
implicit object C :: Format t ⇒ Format (Char → t)
```

This would still allow us to construct format specifiers explicitly (using the dot notation), but it could also be used to construct these implicitly —see Oliveira and Gibbons (2005) for a simple application of this.

3.2 Abstract Datatypes and Bulk Types

This paper does not present a solution for the *expression problem* (Wadler 1998): we cannot use case analysis or pattern matching to define new functions on open datatypes like *Format* in Figure 5. While in some situations this can be a curse, the fact is that, in other situations, this is a blessing: *the lack of a mechanism that inspects the structure of an object ensures encapsulation*. This is essential for ADTs.

In Section 2 we have already seen an example of a *Set* ADT. In that example, we defined a *Set* interface using a class and *Set* implementations using an object. In Figure 6 we can see another example of a very simplified *Graph* ADT, inspired by one of the examples in Chakravarty et al. (2005b). Again, the same idea applies: we define an abstract interface with the possible operations on graphs (in this case we just have *outEdges*) as a class; and we create different implementations of that interface using objects.

It is interesting to compare our approach to ADTs with the related problem of *bulk types* (Peyton Jones 1996) in Haskell. Peyton Jones proposed the use of *constructor classes* (Jones 1993) and *multiple parameter type classes* to solve the problem of bulk types. However, this approach is unecessarelly restrictive, as pointed out in Jones (2000), because it requires constructors that can be written in only a certain form. For example, the constructor class approach would not work for our two implementations of graphs. Functional dependencies and, more recently, associated types solve this problem and allow both implementations of graphs. We show the solution with associated types next.

```
class GraphOps g where
  type Node g
  outEdges :: g → Node g → [(Node g, Node g)]
```

While both *GraphOps* and *Graph* can be seen as alternative solutions to the bulk types problem, they are not equivalent and one approach might be preferable to the other in different scenarios. A solution like *GraphOps* is useful because it provides overloaded operations for multiple graph implementations and, since there is not abstraction, it is possible to use those operations in combination with implementation specific operations (which may have, for example, performance advantages). On the other hand, the ADT solution is useful because it hides the concrete implementation and allows an implementation to be replaced by another one transparently, which has significant advantages from a software engineering point of view. In summary, in the ADT solution *Graph a* can be viewed as an actual parametrized datatype/container, while the *GraphOps g* solution provides (type) overloaded operations for some graph implementation g.

```
insert :: Ord a ⇒ a → [a] → [a]
insert x []       = [x]
insert x (y : ys) =
  if x > y then y : insert x ys else x : y : ys

sort :: Ord a ⇒ [a] → [a]
sort = foldr insert []

sortImplicit :: [Int]
sortImplicit = sort [2, 6, 5]

object OrdReverse :: Ord Int
object OrdReverse where
  x > y = ¬ (primIntGT x y)

sortExplicit :: [Int]
sortExplicit = OrdReverse.sort [2, 6, 5]
```

Figure 7: No need for 'By' functions.

3.3 Explicit Implicit Objects

In the Haskell libraries there is often the need to provide two similar definitions: one for convenience, since it takes a dictionary argument implicitly; and another for flexibility taking an explicit extra argument for the user to provide. We show a (slightly simplified) example taken from the *Data.List* Haskell libraries next:

```
insert :: Ord a ⇒ a → [a] → [a]
...
insertBy :: (a → a → Bool) → a → [a] → [a]
...
sortBy :: (a → a → Bool) → [a] → [a]
sortBy gt = foldr (insertBy gt) []

sort :: Ord a ⇒ [a] → [a]
sort = sortBy (>)
```

Here, the idea is that the *insert* and *sort* functions can be used by the programmer without any need to worry about which comparison function is going to be used. The comparison function is defined in the instance of *Ord* for the element types of the list and it serves the purpose most of the times. There are, however, certain situations where the programmer may be interested to use a different comparison function (for example, if he wants to sort a list in descending order, rather than ascending order). In those situations it is not possible to use *insert* and *sort* because, in Haskell, we cannot explicitly pass a dictionary value. To alleviate that problem, the Haskell libraries often provide a second function taking an extra argument that can be used by the programmer. For example, *insertBy* and *sortBy* are the more flexible versions of *insert* and *sort*. However this solution is not completely satisfactory due to the required duplication.

We can provide a better solution in our system because we can use the dot notation to override an implicit object and we have no need to define the extra 'By' functions. In Figure 7 we show how this can be done and give two examples where we use the *sort* function with an implicit and an explict object. The list *sortImplicit* uses the *Ord Int* object that is on scope at the moment to sort the elements; while the list *sortExplicit* overrides the object on scope and uses *OrdReverse* instead. In essence all the code is essentially the same code that we would have written

```
sealed class Exp a where
object Lit  :: Int → Exp Int
object Plus :: Exp Int → Exp Int → Exp Int
object IsZ  :: Exp Int → Exp Bool
object If   :: Exp Bool → Exp a → Exp a → Exp a

eval              :: Exp a → a
eval (Lit x)       = x
eval (Plus e1 e2)  = eval e1 + eval e2
eval (IsZ e)       = eval e ≡ 0
eval (If p e1 e2)  = if eval p then eval e1 else eval e2
```

Figure 8: A Sealed Class (or GADT) for Typed Expressions.

in Haskell. It is only when we want to explicitly provide an *Ord* object, that we use the dot notation to pass the object.

3.4 A Poor Man's Approach to Implicit Parameters

In Section 3.3 we have shown how to define functions that can take objects either implicitly or explicitly. Our next example shows how we can, more generally, view this as a poor man's approach to *implicit parameters* (or *dynamic scoping*) (Lewis et al. 2000). We demonstrate this by using the main motivating example of Lewis et. al. of a pretty printing function.

```
pretty :: Doc → String
```

In that example, buried somewhere inside the code, we could have:

```
... if i ⩾ 78 then ...
```

This code could be defined 5 levels deep in the recursion so, if we wanted to replace 78 by something more general, we would normally need to either define a global name or add an extra parameter to nearly every function involved in the pretty printing code. However, with our more general notion of classes, we have an alternative option that we show next:

```
class PPArgs where
  width :: Int
...
... if i ⩾ width then ...
```

The idea is that we add a new class *PPArgs* that defines the optional parameters of the pretty printing function (in this case we only have *width*) and we use those in the definitions involved in the pretty printer. Note that, unlike with Haskell type classes, we can have classes with no type arguments. By using *width* instead of 78 our pretty printer would have the type:

```
pretty :: PPArgs ⇒ Doc → String
```

We can make the default width 78 by creating an implicit object for *PPArgs* as follows:

```
implicit object DefPP where width = 78
```

Now, any calls to pretty without explicitly passing a *PPArgs* object would use 78 as the *width*. If we want to use a *width* of 90, all we need to do is to create a new *PPArgs* object:

```
object NightyPP where width = 90
```

and call *NightyPP.pretty*. Although this approach is, perhaps, not as direct as using Lewis et al. (2000) proposal, it does offer a cheap alternative. Furthermore, unlike with implicit parameters, this approach does not have integration issues with classes: we can use *PPArgs* as any other class constraint.

3.5 GADTs as Sealed Classes

As we have seen in Section 2.4, we can use sealed classes to define datatypes and functions by pattern matching in a similar way as in Haskell. However, the **object** syntax can be used to define signatures for data constructors that are more general than the ones found in Haskell 98, effectively allowing us to define GADTs (Peyton Jones et al. 2006). We show a standard GADT example for typed expressions in Figure 8 and define the corresponding evaluation function. In the first line we declare a sealed class for expressions with no methods. In the next four lines (the **object** declarations) we use the syntatic sugar introduced in Section 2.4 to write the types of the object constructors in much the same way we would do with Haskell's GADTs.

The evaluation function is written exactly in the same way that we would write it in Haskell. We should emphasize that, if *Exp* would not be marked as **sealed** and we would try to define *eval* using pattern matching, then we would get a compile-time error saying that since *Exp* is not marked as sealed, we could not use pattern matching.

3.6 Sealed Classes with Methods

In Section 3.5 we have seen how sealed classes allow us to write programs in much the same way as with Haskell's GADTs. Nevertheless, the fact is that sealed classes are more general than GADTs because they can have methods, which seems to be related to *attribute grammars* (Knuth 1968).

Suppose that we extend and develop the example in Figure 8 further to define a full-compiler with parsing, a run-time system, several transformation steps, etc. In that system there will be many functions defined by pattern matching and, for a more realistic language, there may exist several dozens of objects. At some stage, we decide to report better messages to the user, so we try to add extra information about the location of an expression in the source code. For simplicity, we represent that location by a pair of integers representing the line and column:

```
type Loc = Maybe (Int, Int)
```

Now we are faced with the task of adding the extra location information to our program, but how can this be achieved? One solution is to decorate all the recursive occurrences of Exp with the extra location information:

```
type LExp a = (Loc, Exp a)
...
object Plus :: LExp a → LExp a → Exp a
object IsZ  :: LExp Int → Exp Bool
object If   :: LExp Bool → LExp a → LExp a → Exp a
...
```

(A similar approach is used, for example, in the code for the GHC compiler.) However, this involves, basically, touching every single object. Much worse than that, it also involves changing every single definition that uses pattern matching over expressions, forcing the programmer to practically touch all parts of the program. Certainly a tremendous implementation effort! The fact is that, in Haskell, if someone did not envision this scenario in the first place, he would be very likely have to go through it.

Although this scenario looks quite daunting in Haskell, methods come to rescue in our system. Instead of adding the location information to the objects, we add that information directly to the class itself using a method. We show how this can be done in Figure 9. The striking thing to note is that little has changed (when comparing this code with the one in Figure 8) except for the addition of the method *loc* and corresponding definitions in the objects. The function *eval*, for example, remains untouched. In essence the only code that would need some immediate adjustment would be the code related to parsing, which should fill in the right locations in the expressions. With this solution, *code that does not need the extra location information does not need to be changed*. Still, if we add support for division in to our expressions

```
sealed class Exp a where
...
object Div :: Exp Int → Exp Int → Exp Int
object Div e1 e2 where loc = e1.loc
```

```
sealed class Exp a where
  loc :: Loc

object Lit   :: Int → Exp Int
object Lit where loc = Nothing

object Plus :: Exp Int → Exp Int → Exp Int
object Plus e1 e2 where loc = e1.loc

object IsZ  :: Exp Int → Exp Bool
object IsZ e where loc = e.loc

object If    :: Exp Bool → Exp a → Exp a → Exp a
object If e1 e2 e3 where loc = e1.loc

eval               :: Exp a → a
eval (Lit x)       = x
eval (Plus e1 e2)  = eval e1 + eval e2
eval (IsZ e)       = eval e ≡ 0
eval (If p e1 e2)  = if eval p then eval e1 else eval e2
```

Figure 9: Typed Expressions with Extra Location Information.

we can make use of the location information when reporting a run-time ‘division by zero’ error by just looking up the location information for that expression. Here is how we could modify *eval* to do this:

```
eval :: Exp a → a
...
eval d@(Div e1 e2) =
  if eval e2 ≢ 0 then eval e1 ‘div‘ eval e2
  else error (show d.loc ++ ": division by 0")
```

As a final remark, note that the method $loc :: Exp\ a \Rightarrow Loc$ would be ambiguous in Haskell. Variable a does not appear in the type and therefore there is an ambiguity if we pass dictionaries implicitly. However, because we can explicitly pass objects, ambiguity in our system is not problematic.

4. Core Calculus

We define an extension of Hindley/Milner with object-style classes, which additionally support the explicit manipulation of evidence (dictionaries). Section 4.1 introduces the syntax of programs and Section 4.2 explains the meaning of programs by applying the classic dictionary translation (Hall et al. 1996). We will postpone the treatment of sealed classes until the later Section 4.3. The issue of type inference and context reduction (automatic resolution/construction of objects/dictionaries) is left for future work.

4.1 Syntax of Programs

First, we consider the syntax of programs, which is given in Figure 10. The significant deviation from type classes is that we view class symbols C as a form of type constructor. Thus, type terms such as `Eq (Eq a)` are well-formed in our system. In each context, we attach variable names x_i to types t_i in a (type) context. The idea is that x_i refers to the dictionary connected to the type t_i. Consequently, we can explicitly refer to dictionaries in the program text.

The expression language is standard. The new construct $e\#e$ allows the user to provide evidence in the form of a dictionary for a context. Via # we can express the earlier ”dot” notation in a more primitive form. For example, $x.m$ is a shorthand for $m\#x$. Class declarations are exactly like in the type class case and allow us to group together related methods (though we only consider a single method for simplicity). Instead of type class instance declarations we have now object declarations. An implicit object behaves exactly like a type class instance declaration. We will generate a proof rule from the declaration, which can be used to automatically derive type classes. In addition, we attach a constructor name K to each declaration. The constructor K refers to the dictionary (function) generated from the declaration (and thus the user can build explicit dictionaries in source programs). For standard type classes, the instance context only holds type classes. But we consider classes as types. Hence, we allow for a wider range of functions that build dic-

Type names	s, t, u, v
Variable names	x, y, z, a, b

Types

t	$::=$	x	Variables
	$\mid$	$t \rightarrow t$	Functions
	$\mid$	$(t, ..., t)$	Products
	$\mid$	$C\ t...t$	Classes
	$\mid$	$ctx \Rightarrow t$	Context types
σ	$::=$	$t \mid \forall\bar{a}.t$	Type schemes

Context

$ctx \quad ::= \quad (x_1 : t_1, ..., x_n : t_n)$

Expressions

e	$::=$	$x \mid K$	Variables and constructors
	$\mid$	$\lambda x.e \mid e\ e$	Abstraction and application
	$\mid$	$\mathsf{let}\ x = e\ \mathsf{in}\ e$	Let definition
	$\mid$	$(e :: \sigma)$	Annotation
	$\mid$	$(e_1, ..., e_n)$	Products
	$\mid$	$e\#e$	Explicit context

Declarations

$$\begin{array}{lll} decl & ::= & \mathsf{class}\ C\ a_1...a_n \mathsf{where}\ m :: ctx \Rightarrow t \\ & \mid & \mathsf{implicit\ object}\ K :: ctx \Rightarrow C\ t_1...t_n\ \mathsf{where}\ m = e \\ & \mid & \mathsf{object}\ K x_1...x_m :: \\ & & \quad ctx \Rightarrow t'_1 \rightarrow ... \rightarrow t'_m \rightarrow C\ t_1...t_n\ \mathsf{where}\ m = e \end{array}$$

Environment

v	$::=$	$x \mid K$	
Γ	$::=$	$\{v : \forall\bar{a}.ctx \Rightarrow t\}$	Type assignment
	$\mid$	$\Gamma \cup \Gamma$	
Γ_R	$::=$	$\{K : \forall\bar{a}.ctx \Rightarrow t\}$	Proof rule
	$\mid$	$\Gamma_R \cup \Gamma_R$	

Substitutions

$\theta \quad ::= \quad [t_1/a_1, ..., t_n/a_n]$

Target

$$\begin{array}{lll} T & ::= & x \mid T \rightarrow T \mid (T, ..., T) \mid C\ T...T \mid \forall\bar{a}.T \\ E & ::= & x \mid k \mid \lambda x : T.E \mid E\ E \\ & \mid & \Lambda a.E \mid E\ T \mid \mathsf{let}\ x = E\ \mathsf{in}\ E \end{array}$$

Shorthands

$() \Rightarrow t \equiv t$

$\overline{a} \equiv a_1, ..., a_n$

$\overline{t} \equiv t_1...t_n$

$\overline{x : t} \equiv x_1 : t_1, ..., x : n : t_n$

$[\overline{t/a}] \equiv [t_1/a_1, ..., t_n/a_n]$

$\forall.ctx \Rightarrow t \equiv \forall tv(ctx, t).ctx \Rightarrow t$

$\Lambda\bar{a}. \equiv \Lambda a_1...\Lambda a_n.$

$tv :: Term \rightarrow FreeVariables$

Figure 10: Syntax of Programs

$$\boxed{\overline{decl} \rightsquigarrow (\Gamma_R, \Gamma)}$$

$$\text{(Decls)}\ \frac{decl_i \rightsquigarrow (\Gamma_{P_i}, \Gamma_i) \quad i = 1, ..., n}{decl_1, ..., decl_n \rightsquigarrow (\Gamma_{P_1} \cup ... \cup \Gamma_{P_n}, \Gamma_1 \cup ... \cup \Gamma_n)} \qquad \text{(Cls)}\ \frac{x \text{ fresh}}{\begin{array}{c}\textsf{class}\ C\ a_1...a_n \textsf{where}\ m :: (x_1 : t_1, ..., x_k : t_k) \Rightarrow t \rightsquigarrow \\ (\{\}, \{m : \forall.(x : C\ a_1...a_n, x_1 : t_1, ..., x_k : t_k) \Rightarrow t\})\end{array}}$$

$$\text{(OImpl)}\ \frac{\Gamma = \{K : \forall.ctx \Rightarrow C\ \bar{t} \quad \}}{\textsf{implicit object}\ K :: ctx \Rightarrow C\ \bar{t}\ \textsf{where}\ m = e \rightsquigarrow (\Gamma, \Gamma)} \qquad \text{(OExpl)}\ \frac{\Gamma = \{K : \forall.ctx \Rightarrow t\}}{\textsf{object}\ K\bar{x} :: ctx \Rightarrow t\ \textsf{where}\ m = e \rightsquigarrow (\{\}, \Gamma)}$$

$$\boxed{\overline{decl} \rightsquigarrow \Gamma_{Target} \cup DataDecl_{Target}}$$

$$\text{(Obj)} \qquad [\textsf{implicit}]\ \textsf{object}\ K\bar{x} :: (y_1 : t_1, ..., y_n : t_n) \Rightarrow t\ \textsf{where}\ m = e \rightsquigarrow k : \forall.t_1 \to ... \to t_n \to t$$

$$\text{(ClsToData)}\ \frac{\bar{b} = \mathit{tv}(t_1, ..., t_k) - \mathit{tv}(a_1, ..., a_n)}{\textsf{class}\ C\ a_1...a_n \textsf{where}\ m :: (x_1 : t_1, ..., x_k : t_k) \Rightarrow t \rightsquigarrow \textsf{data}\ C\ a_1...a_n = CC\{m :: \forall\bar{b}.t_1 \to ... \to t_k \to t\}}$$

Figure 11: Source and target environment generation

tionaries. For example, the constructor K may take additional arguments x_i, besides the (dictionary) arguments in the context. This is only possible for "non-implicit" object declarations. For such declarations we will not generate a proof rule, effectively disallowing automatic inference.

The environment Γ contains the types of built-in functions, lambda-/let-bound variables and dictionary functions K. In the target language we will use a lower-case k, which is the common notation for function names. For proof rules we have a second environment Γ_R. We use type assignment notation for proof rules. For example, consider the familiar case of $EqList : \forall a.Eq\ a \Rightarrow Eq\ [a]$. We can built a proof (dictionary) for $Eq\ [a]$ provided we have a proof for $Eq\ a$. We assume that the environments Γ and Γ_R are well-formed. That is, there are no two conflicting assignments $x : \sigma_1$ and $x : \sigma_2$ in Γ (as well as in Γ_R) such that σ_1 and σ_2 differ. Substitutions map variables to types and arise when building instances of types. The target type and expression language is the familiar one from the type class case: System F extended with datatypes.

4.2 Type-directed Translation of Programs

We describe the meaning of programs by applying the classic dictionary translation. Like for the standard type class case, each class declaration

$$\textsf{class}\ C\ a_1...a_n \textsf{where}\ m :: (x_1 : t_1, ..., x_k : t_k) \Rightarrow t$$

translates to a datatype declaration

```
data C a1...an = CC {m :: ∀b̄. t1 →...→ tk → t}
```

where $\bar{b} = \mathit{tv}(t_1, ..., t_k) - \mathit{tv}(a_1, ..., a_n)$. For convenience, we use the labelled datatype notation, which will make the translation of methods straightforward.

In the context of the method's declaration, variables are attached to types to aid the dictionary-translation process. Values belonging to the above datatype are referred to as dictionaries. They can be viewed as a proof (evidence) that a method definition is defined for an instance of the class. As we will see shortly, the translation of object declarations to dictionary functions is almost identical to the standard type class case. But we are more general because we allow types in a context and the dictionary function can take extra arguments. In contrast to Haskell type classes, we do *not* impose the unambiguity condition that the variables $a_1, ..., a_n$ and the variables in $t_1, ..., t_k$ must appear in t. The reason is that in our setting we can easily deal with (potentially) ambiguous types by passing dictionaries explicitly. See the earlier example in Section 3.6.

Figure 11 contains rules for generating the source and target type environments. For example, judgements $\overline{decl} \rightsquigarrow (\Gamma_R, \Gamma)$ compute the types of methods and dictionary functions from a sequence of class and object declarations recorded in Γ and the proof rules recorded in Γ_R. The point to note is that implicit object declarations yield a type assignment and a proof rule. We need both environments to translate source expressions to target expressions. The translation of source declarations and expressions is given in Figure 12.

Proof rules describing the translation of expressions make use of judgements $\Gamma_R, \Gamma \vdash e : \sigma \rightsquigarrow E$ where Γ_R is the proof rule environment and Γ is the type assignment environment, e the source expression, σ the source type and E the target expression. All of the rules, with the exception of rule (CtxtExpl), are the familiar dictionary-translation rules also employed for the translation of standard type classes. Rule (Var) deals with lambda-/let-bound variables whereas rule (Var-K) deals with dictionary constructors. The rules (Abs), (App), (Let), (Annot) and (Product) for function abstraction/application, let-definitions, type annotations and products, as well as the rules ($\forall$Intro) and ($\forall$Elim) for quantifier introduction and elimination, are straightforward. Rule (CtxtIntro) deals with introduction of the type context, which is turned into an explicit dictionary argument in the translation. We should note that, in our system, a context is nothing else than a (proof rule) environment. Recall that classes are types and are always attached to a dictionary variable. The rule (CtxtImpl) deals with the (implicit) elimination of the context. We need to build dictionaries E_i for the types t_i in the context. These dictionaries are implicit. That is, they must be inferred, which is indicated by the fact that the source (dictionary) expressions e_i are not part of the program text e and we can only make use of the proof rule environment Γ_R to generate those e_i. In our system, we provide the user with the ability to explicitly provide dictionaries. See rule (CtxtExplicit) where the argument e' plays the role of the e_i's.

We yet need to process object declarations to give meaning to the dictionary functions K. This step is performed via rules (Decls) and (O). Rule (O) deals with implicit as well as non-implicit declarations indicated by the optional keyword [implicit].

$$\boxed{\Gamma_R, \Gamma \vdash e : \sigma \leadsto E}$$

$$\text{(Var)}\ \frac{(x : \sigma) \in \Gamma \cup \Gamma_R}{\Gamma_R, \Gamma \vdash x : \sigma \leadsto x} \qquad \text{(Abs)}\ \frac{\Gamma_R, \Gamma \cup \{x : t_1\} \vdash e : t_2 \leadsto E}{\Gamma_R, \Gamma \vdash \lambda x.e : t_1 \to t_2 \leadsto \lambda x : t_1.E} \qquad \text{(App)}\ \frac{\begin{array}{c}\Gamma_R, \Gamma \vdash e_1 : t_2 \to t_1 \leadsto E_1 \\ \Gamma_R, \Gamma \vdash e_2 : t_2 \leadsto E_2\end{array}}{\Gamma_R, \Gamma \vdash e_1\ e_2 : t_1 \leadsto E_1\ E_2}$$

$$\text{(Var-K)}\ \frac{(K : \sigma) \in \Gamma \cup \Gamma_R}{\Gamma_R, \Gamma \vdash K : \sigma \leadsto k}$$

$$\text{(Let)}\ \frac{\Gamma_R, \Gamma \vdash e_1 : \sigma \leadsto E_1 \quad \Gamma_R, \Gamma \cup \{x : \sigma\} \vdash e_2 : t \leadsto E_2}{\Gamma_R, \Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : t \leadsto \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2} \qquad \text{(Annot)}\ \frac{\Gamma_R, \Gamma \vdash e : \sigma \leadsto E}{\Gamma_R, \Gamma \vdash (e :: \sigma) : \sigma \leadsto E}$$

$$\text{(}\forall\text{Intro)}\ \frac{\Gamma_R, \Gamma \vdash e : t \leadsto E \quad \bar{a} = \mathit{tv}(t) - \mathit{tv}(\Gamma \cup \Gamma_R)}{\Gamma_R, \Gamma \vdash e : \forall \bar{a}.t \leadsto \Lambda \bar{a}.E} \qquad \text{(}\forall\text{Elim)}\ \frac{\Gamma_R, \Gamma \vdash e : \forall \bar{a}.t' \leadsto E}{\Gamma_R, \Gamma \vdash e : [\overline{t/a}]t' \leadsto E\ \bar{t}}$$

$$\text{(CtxtIntro)}\ \frac{\Gamma_R \cup \{x_1 : t_1, ..., x_n : t_n\}, \Gamma \vdash e : t' \leadsto E}{\begin{array}{c}\Gamma_R, \Gamma \vdash (x_1 : t_1, ..., x_n : t_n) \Rightarrow t' \leadsto \\ \lambda x_1 : t_1 ... \lambda x_n : t_n.E\end{array}} \qquad \text{(CtxtImpl)}\ \frac{\begin{array}{c}\Gamma_R, \Gamma \vdash e : (x_1 : t_1, ..., x_k : t_k) \Rightarrow t \leadsto E \\ \Gamma_R, \{\} \vdash (e_1, ..., e_k) : (t_1, ..., t_k) \leadsto (E_1, ..., E_k)\end{array}}{\Gamma_R, \Gamma \vdash e : t \leadsto E\ E_1 ... E_k}$$

$$\text{(Product)}\ \frac{\Gamma_R, \Gamma \vdash e_i : t_i \leadsto E_i \quad x_i\ \text{fresh for}\ i = 1, ..., n}{\Gamma_R, \Gamma \vdash \begin{array}{c}(e_1, ..., e_n) : (x_1 : t_1, ..., x_n : t_n) \\ \leadsto \\ (E_1, ..., E_n)\end{array}} \qquad \text{(CtxtExpl)}\ \frac{\begin{array}{c}\Gamma_R, \Gamma \vdash e : (x_1 : t_1, ..., x_k : t_k) \Rightarrow t \leadsto E \\ \Gamma_R, \Gamma \vdash e' : (t_1, ..., t_k) \leadsto (E_1, ..., E_k)\end{array}}{\Gamma_R, \Gamma \vdash e \# e' : t \leadsto E\ E_1 ... E_k}$$

$$\boxed{\Gamma_R, \Gamma \vdash \overline{odecls} \leadsto \overline{k = E}}$$

$$\text{(Decls)}\ \frac{\Gamma_R, \Gamma \vdash odecl_i \leadsto k_i = E_i \quad \text{for}\ i = 1, .., n}{\Gamma_R, \Gamma \vdash odecl_1, ..., odecl_n \leadsto k_1 = E_1, ..., k_n = E_n}$$

$$\text{(O)}\ \frac{\begin{array}{c}(m : \forall \bar{a}, \bar{c}.(x : C\ a_1 ... a_n, x_1 : t'_1, ..., x_k : t'_k) \Rightarrow t) \in \Gamma \\ \theta = [t_1/a_1, ..., t_n/a_n] \\ \bar{b} = \mathit{tv}(s_1, ..., s_m, t) - \mathit{tv}(\Gamma) \quad \bar{c} = \mathit{tv}(t'_1, ..., t'_k) - \mathit{tv}(a_1, ..., a_n) \\ \Gamma_R, \Gamma \vdash e : \forall \bar{b}, \bar{c}.(y_1 : s_1, ..., y_m : s_m, x_1 : \theta(t'_1), ..., x_k : \theta(t'_k)) \Rightarrow \theta(t) \leadsto E\end{array}}{\Gamma_R, \Gamma \vdash \begin{array}{c}[\mathsf{implicit}]\ \mathsf{object}\ K z_1 ... z_l :: (y_1 : s_1, ..., y_m : s_m) \Rightarrow u_1 \to ... \to u_l \to C\ t_1 ... t_n\ \mathsf{where}\ m = e \\ \leadsto \\ k = \Lambda \bar{b}.\lambda y_1 : s_1 ... \lambda y_m : s_m.\lambda z_1 : u_1 ... \lambda z_l : u_l.CC\ t_1 ... t_n\ (\Lambda \bar{c}.\lambda x_1 : \theta(t'_1) ... \lambda x_k : \theta(t'_k).E)\end{array}}$$

Figure 12: Type-directed translation rules

In case $l = 0$, rule (O) is effectively equivalent to the standard type class instance translation step. Our task is to build a dictionary function that, given some dictionaries for the object context, builds a dictionary for $C\ t_1 ... t_n$. We first translate the method body under the instantiated type by replacing a_i's by t_i's. Variables $\bar{c}$ refer to all the remaining (free) variables in the declared type of the method and variables $\bar{b}$ are all those variables not bound by the environment. Hence, we can universally quantify over $\bar{b}$ and $\bar{c}$. In the type context, we find $y_1 : s_1, ..., y_m : s_m$ from the object declaration context and $x_1 : \theta(t'_1), ..., x_k : \theta(t'_k)$ from the class declaration context. We can refer to dictionary variables y_i's and x_j's in the program text of e. The dictionary function K is built by abstracting over the set of free type variables $\bar{b}$ and abstracting over the dictionary variables y_i from the object context. The dictionary for $C\ t_1 ... t_n$ is built by applying the class constructor CC to the types t_i followed by the application to $(\Lambda \bar{c}.\lambda x_1 : \theta(t'_1) ... \lambda x_k : \theta(t'_k).E)$, which effectively represents the actual dictionary definition. The type abstraction over $\bar{c}$ captures the ”locally” quantified variables and the function abstraction over x_j the ”locally” provided dictionaries.

It is straightforward to verify that the translation rules yield well-typed target expressions. Thus, we achieve soundness of our system.

4.3 Sealed Classes Extension

We consider the extension to sealed classes. The syntax and translation rules of the sealed classes extensions are given in Figure 13. The main difference is that each object declaration yields a constructor of the sealed class data type. See rule (SClsToData) where, for simplicity, we only consider the case of a single object declaration. In the target language, the constructor K takes the method definition, the context and the parameters z_i as arguments. The output type $C\ v_1 ... v_n$ may be of a more specialized type than the (data) declaration. Hence, we actually need GADTs in the target language. We omit the details how to translate GADTs into a more foundational calculus such as for example System F_C (Sulzmann

Expressions

$$\begin{array}{lll} e & ::= & \ldots \mid \mathsf{case}\ e\ \mathsf{of}\ [p_i \rightarrow e_i]_{i \in I} \quad \text{Case} \\ p & ::= & K\ x_1 \ldots x_n \quad \text{Pattern} \end{array}$$

Declarations

$$decl \quad ::= \quad \ldots \mid \mathsf{sealed\ class}\ C\ a_1 \ldots a_n \mathsf{where}\ m :: ctx \Rightarrow t$$

Target

$$\begin{array}{lll} E & ::= & \ldots \mid \underline{\ }K \mid \mathsf{case}\ E\ \mathsf{of}\ [P_i \rightarrow E_i]_{i \in I} \\ P & ::= & K\ \bar{t}\ x_1 \ldots x_n \end{array}$$

$$\boxed{\overline{decl} \rightsquigarrow (\Gamma_R, \Gamma)}$$

$$\text{(SCls)} \quad \frac{x\ \text{fresh}}{\begin{array}{c}\mathsf{sealed\ class}\ C\ a_1 \ldots a_n \mathsf{where}\ m :: (x_1 : t_1, \ldots, x_k : t_k) \Rightarrow t \rightsquigarrow \\ (\{\}, \{m : \forall.(x : C\ a_1 \ldots a_n, x_1 : t_1, \ldots, x_k : t_k) \Rightarrow t\})\end{array}}$$

$$\boxed{\overline{decl} \rightsquigarrow \Gamma_{Target} \cup DataDecl_{Target}}$$

$$\text{(SClsToData)} \quad \frac{\begin{array}{c}\bar{b} = \mathit{tv}(t_1, \ldots, t_k) - \mathit{tv}(a_1, \ldots, a_n) \\ [\mathsf{implicit}]\ \mathsf{object}\ K\ z_1 \ldots z_l :: (y_1 : s_1, \ldots, x_m : s_m) \Rightarrow u_1 \rightarrow \ldots \rightarrow u_l \rightarrow C\ v_1 \ldots v_n\end{array}}{\begin{array}{c}\mathsf{sealed\ class}\ C\ a_1 \ldots a_n \mathsf{where}\ m :: (x_1 : t_1, \ldots, x_k : t_k) \Rightarrow t \\ \rightsquigarrow \\ \begin{array}{l}\mathsf{data}\ C\ a_1 \ldots a_n\ \mathsf{where} \\ \quad K :: (\forall \bar{b}.t_1 \rightarrow \ldots \rightarrow t_k \rightarrow t) \rightarrow (s_1, \ldots, s_m) \rightarrow u_1 \rightarrow \ldots \rightarrow u_l \rightarrow C\ v_1 \ldots v_n \\ \\ m :: \forall \bar{a}.C\ a_1 \ldots a_n \rightarrow \forall \mathit{tv}(t_1, \ldots, t_k) - \mathit{tv}(a_1, \ldots, a_n).t_1 \rightarrow \ldots \rightarrow t_k \rightarrow t \\ m\ (K\ mm\ s1n\ z_1 \ldots z_l) = mm\end{array}\end{array}}$$

$$\boxed{\Gamma_R, \Gamma \vdash e : \sigma \rightsquigarrow E}$$

$$\text{(Case)} \quad \frac{\begin{array}{c}\Gamma_R, \Gamma \vdash e : t \rightsquigarrow E \\ \Gamma_R, \Gamma \vdash p_i \rightarrow e_i : t \rightarrow t' \rightsquigarrow P_i \rightarrow E_i \quad \text{for } i = 1, \ldots, n\end{array}}{\Gamma_R, \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ [p_i \rightarrow e_i]_{i \in I} : t' \rightsquigarrow \mathsf{case}\ E\ \mathsf{of}\ [P_i \rightarrow E_i]_{i \in I}}$$

$$\text{(Pat)} \quad \frac{\begin{array}{c}K :: \forall \bar{a}.(\forall \bar{b}.t_1 \rightarrow \ldots \rightarrow t_k \rightarrow t) \rightarrow (s_1, \ldots, s_m) \rightarrow u_1 \rightarrow \ldots \rightarrow u_l \rightarrow C\ v_1 \ldots v_n \\ \bar{a} \cap \mathit{tv}(\Gamma_R, \Gamma, v'_1, \ldots, v'_n, t') = \{\} \quad \bar{c} = \bar{a} \cap \mathit{tv}(v_1, \ldots, v_n) \quad \phi = [\overline{t'/c}] \quad \phi(v_1) = v'_1 \ldots \phi(v_n) = v'_n \\ \Gamma_R \cup \{y_1 : \phi(s_1), \ldots, y_m : \phi(s_m)\}, \Gamma \cup \{z_1 : \phi(u_1), \ldots, z_l : \phi(u_l)\} \vdash e : t' \rightsquigarrow E\end{array}}{\Gamma_R, \Gamma \vdash K\ z \ldots z_l \rightarrow e : C\ v'_1 \ldots v'_n \rightarrow t' \rightsquigarrow K\_\ (y_1, \ldots, y_m)\ z_1 \ldots z_l \rightarrow E}$$

$$\boxed{\Gamma_R, \Gamma \vdash \overline{odecls} \rightsquigarrow \overline{k = E}}$$

$$\text{(SO)} \quad \frac{\begin{array}{c}(m : \forall.\bar{a}, \bar{c}.(x : C\ a_1 \ldots a_n, x_1 : t'_1, \ldots, x_k : t'_k) \Rightarrow t) \in \Gamma \quad C\ a_1 \ldots a_n\ \text{is sealed} \\ \theta = [t_1/a_1, \ldots, t_n/a_n] \\ \bar{b} = \mathit{tv}(s_1, \ldots, s_m, t) - \mathit{tv}(\Gamma) \quad \bar{c} = \mathit{tv}(t'_1, \ldots, t'_k) - \mathit{tv}(a_1, \ldots, a_n) \\ \Gamma_R, \Gamma \vdash e : \forall \bar{b}, \bar{c}.(y_1 : s_1, \ldots, y_m : s_m, x_1 : \theta(t'_1), \ldots, x_k : \theta(t'_k)) \Rightarrow \theta(t) \rightsquigarrow E\end{array}}{\Gamma_R, \Gamma \vdash \begin{array}{c}[\mathsf{implicit}]\ \mathsf{object}\ K z_1 \ldots z_l :: (y_1 : s_1, \ldots, y_m : s_m) \Rightarrow u_1 \rightarrow \ldots \rightarrow u_l \rightarrow C\ t_1 \ldots t_n\ \mathsf{where}\ m = e \\ \rightsquigarrow \\ k = \Lambda \bar{b}.\lambda y_1 : s_1 \ldots \lambda y_m : s_m.\lambda z_1 : u_1 \ldots \lambda z_l : u_l.K\ t_1 \ldots t_n\ (\Lambda \bar{c}.\lambda x_1 : \theta(t'_1) \ldots \lambda x_k : \theta(t'_k).E)\ (y_1, \ldots, y_m)\ z_1 \ldots z_l\end{array}}$$

Figure 13: Sealed classes extension

Example:

$\textbf{sealed class}\ Eq\ a\ \textbf{where}$
$\quad (\equiv) :: a \to a \to Bool$

$\textbf{implicit object}\ EqList :: Eq\ a \Rightarrow Eq\ [a]\ \textbf{where}$
$\quad (\equiv)\ [\,]\ [\,] \qquad\qquad\quad = True$
$\quad (\equiv)\ (x : xs)\ (y : ys) = (x \equiv y) \wedge (xs \equiv ys)$
$\quad (\equiv)\ \_\ \_ \qquad\qquad\quad\ = False$

$f :: Eq\ a \to a \to a \to Bool$
$f\ EqList\ xs\ ys = (\equiv)\ xs\ ys$

Translation:

$\textbf{data}\ Eq\ a\ \textbf{where}$
$\quad EqList :: ([a] \to [a] \to Bool) \to Eq\ a \to Eq\ [a]$

$eqMethod :: Eq\ a \to (a \to a \to Bool)$
$eqMethod\ (EqList\ eq\ \_) = eq$

$eqList :: Eq\ a \to Eq\ [a]$
$eqList\ d = EqList\ (\lambda xs \to \lambda ys \to$
$\quad \textbf{case}\ (xs, ys)\ \textbf{of}$
$\qquad ([\,], [\,]) \qquad\quad \to True$
$\qquad (x : xs, y : ys) \to eqMethod\ d\ x\ y\ \wedge$
$\qquad\qquad\qquad\qquad\quad eqMethod\ (eqList\ d)\ xs\ ys$
$\quad)\ d$

$f :: Eq\ a \to a \to a \to Bool$
$f\ (EqList\ \_\ d)\ xs\ ys = eqMethod\ (eqList\ d)\ xs\ ys$

Figure 14: Translation example

et al. 2007). Method application is now also slightly different. We need to pattern match over the possible shapes of dictionaries to access the actual method definition.

The translation of case expressions is performed by rules (Case) and (Pat). For convenience, we assume that arguments of constructors are variables and treat $p \to e$ as an "intermediate" pattern clause expression. Rule (Pat) is the GADT pattern match typing rule (Peyton Jones et al. 2006). To translate the pattern clause in rule (Pat), we translate the pattern body under the extended environment. We must be careful that none of the universally quantified variables $\bar{a}$ (sometimes referred to as existential or abstract variables) escapes. See the side condition $\bar{a} \cap tv(\Gamma_R, \Gamma, v'_1, ..., v'_n, t') = \{\}$. We build the extended environment by computing the type refinement ϕ by matching the provided pattern type against the (output) type of the constructor. Variables z_i appear explicitly in the pattern. They are recorded in the type assignment environment. The type context is implicit and therefore we record the associated dictionary variables y_j in the proof rule environment. The constructor K carries the method definition which does not matter here. Hence, we introduce a do not care pattern variable in the translation.

Object declarations belonging to a sealed class are translated by (SO). The difference to the earlier rule (O) is that we use the declaration specific constructor K instead of the common class constructor CC. The constructor K also takes additional arguments. The type context and the object declaration arguments z_i.

In Figure 14 we consider an example to illustrate the translation process for sealed classes. For brevity, we only consider one object declaration whose behaviour and translation is exactly like in the standard Haskell type class case. Function f pattern matches over Eq's objects. The program text of f demands an implicit $Eq\ [a]$ dictionary which we can implicitly build from the provided $Eq\ a$. See the translation on the right-hand side.

5. Discussion and Related Work

Haskell Type Classes and GADTs The main goal of this paper is to show how, with a single construct, we can generalize and unify both type classes (Hall et al. 1996) and GADTs (Peyton Jones et al. 2006). There are three main ways in which our classes generalize Haskell's (multiple-parameter) type classes. Firstly, objects replace type class instances and allow dictionaries to be built explicitly as well as implicitly (with implicit objects). Type class instances can be view as *anonymous implicit objects*. Secondly, classes are just types and can occur in any type position, while Haskell's type classes can only occur on constraints. Thirdly classes do not need to be parametrized, which is not the case for Haskell's type classes. When compared to Haskell's GADTs, classes can be used to define both open and closed GADTs and they can be passed to a function implicitly or explicitly; in Haskell we can only have closed definitions and values of datatypes are always passed explicitly. Furthermore, in Haskell there is no such concept as GADTs with methods, while our classes can have methods associated with a class.

In essence, the combination of the advantages of GADTs and type classes in a single construct, together with the dot notation (which elegantly allows switching from implicit to explicit parametrization) adds a flexibility and convenience to our system that Haskell does not have.

Omega Propositions Omega (Sheard 2005) has the concept of *propositions*, which stand between Haskell type classes and GADTs: they can be implicitly constructed and passed, but they are closed and have no methods. Propositions can be easily encoded in our system by using a sealed class with no methods where all the objects are implicit objects.

Associated Types and Datatypes Associated types (Chakravarty et al. 2005b) and associated datatypes (Chakravarty et al. 2005a) provide, respectively, a mechanism to declare type synonyms and datatypes that are local to a type class; for different instances we will have different instantiations of the type synonyms or datatypes. While many of the examples using associated types and datatypes have closely related solutions in our system, this is mostly coincidental and we consider the work in this paper to be quite orthogonal that work. In fact, associated types and datatypes can be seen as complementary features that add extra power to our class system. As a short example showing how we could use both features together, consider the problem of defining a (closed) type-indexed function that adds two type-level naturals. Assuming an hypothetical associated types extension we could do so by:

$\textbf{sealed class}\ Add\ m\ n\ \textbf{where}$
$\quad \textbf{type}\ Plus\ m\ n$

$\textbf{implicit object}\ Base :: Add\ Z\ n$
$\textbf{implicit object}\ Base\ \textbf{where}$
$\quad \textbf{type}\ Plus\ Z\ n = n$

$\textbf{implicit object}\ Step :: Add\ m\ n \Rightarrow Add\ (S\ m)\ n$
$\textbf{implicit object}\ Step\ \textbf{where}$
$\quad \textbf{type}\ Plus\ (S\ m)\ n = S\ (Add\ m\ n)$

We could use this type-level addition to define, for example, an *append* function on vectors —that is, lists (type) indexed by their length.

$append :: Add\ m\ n \Rightarrow Vector\ a\ m \to Vector\ a\ n \to$
$\qquad\qquad\qquad\qquad\quad Vector\ a\ (Plus\ m\ n)$

Note that this $append$ function can take values of $Add\ m\ n$ both implicitly or explicitly, which can be useful in different situations: we may be interested in letting the compiler implicitly infer values of $Add\ m\ n$ for us when m and n are known statically; but we can also explicitly build one such value at run-time when we do not have all the information statically.

Named Instances and Explicit Implicit Parameters Kahl and Scheffczyk (2001) propose a language extension to enhance the late binding capabilities of Haskell type classes. The basic idea is to allow named type class instances and use those names to construct and override dictionaries. In their proposal, named instances and Haskell modules share the same name space and are separate from normal values. In order to handle the issues with context reduction, they make the distinction between ordered and unordered of constraints. With ordered constraints no context reduction is performed, so it is possible to get the most of late binding; with unordered constraints dictionaries cannot be overridden and context reduction is performed as in Haskell.

Dijkstra and Swierstra (2005) propose a similar idea but dictionaries are manipulated as a special form of records and they share a namespace with normal values. Furthermore, instead of making a distinction between ordered and unordered constraints, in their approach the programmer explicitly states which instances that should participate in the proof process. This is similar to our approach, which uses implicit objects for that purpose. They also propose that quantifiers and predicates (type class constraints) should be placed in a signature as much to the right as possible in order to retain polymorphism in the type inferencer for as long as possible; and they use partial type signatures to alleviate the burden of explicitly specifying the (full) signatures. Finally, their system handles higher-order predicates, which we decided not to support to keep our system simple. However, we are interested in exploring the addition of higher-order predicates in the future.

There are number of differences between Kahl and Scheffczyk (2001); Dijkstra and Swierstra (2005) and the work proposed in this paper. Firstly, in both approaches, type classes are still second class types (that is, they cannot appear in type positions). With our approach this is not the case and we can use classes at any type position. Secondly, objects can take any kind of arguments as parameters, while instances can only take class dictionaries as arguments. As a consequence, we cannot, in general, define datatypes/container types such as our $Graph$, Set, $Format$ and Exp examples in those approaches—the exception is for datatypes like Omega propositions, where all the parameters on the constructors can be modelled as dictionaries. Thirdly, sealed classes are not considered by them. Finally, because we use standard value constructor syntax and a familiar dot notation, we argue that our source syntax for dictionary construction and application is much more simple and intuitive than theirs.

Haskell object-oriented programming Hughes and Sparud (1995) point out that, although Haskell provides excellent support for writing reusable code, object-oriented style inheritance is not supported. They propose the addition of object classes and object instances (which bear some syntactic similarity to our objects) to solve this problem and allow inheritance in subclasses. In this paper we have not explored subclassing, but we intent to do so in the future and we hope to get the benefits that Hughes and Sparud promote. Nordlander (1999) proposes an object-oriented Haskell-like language for reactive programming. In that system type classes are subsumed by *structs* and there is a separate construct for algebraic datatypes. The language supports subtyping and both structs and datatypes can have subtypes. This makes Nordlander system considerably more complex than what we propose here. Kiselyov and Lämmel (2005) propose an entirely different approach to object-oriented programming in Haskell, by having a library-based approach (instead of a new language or extension) using advanced (and experimental) type class features. While their library shows that a lot of advanced object-oriented programming can be encoded in plain (GHC) Haskell, their approach suffers from not having any compiler support — types are can be very complex and there is no proper syntatical support.

Meijer and Claessen (1997) observe that some of Haskell's constructs are too complex and they propose a similar idea to ours: unifying type classes and datatypes in a single construct. However, their approach is quite different and has typing issues that we do not have. Firstly, they do not have a separate concept of object (that is, all objects are anonymous). This design decision means that hierarchies are modelled solely using classes (in a similar style to conventional OO languages like Java or C#), but in a system without "real" subtyping this can be problematic. For example, even though they can have a $head$ function with type $Cons\ a \to List\ a$, because all type-checking remains covariant, they can still apply $head$ to an empty list, resulting on the same pattern matching error that we would have on Haskell. We avoid this issue in our system by having a separate notion of objects, which does not introduce a new type $Cons\ a$. While we would still get the pattern matching error if we allowed partial functions, with sealed classes we could also take a different design decision and forbid partial functions in the first place, which would rule out these pattern matching errors. Secondly, they can apply pattern matching to any kind of classes, which results in extra "message not understood" errors and breaks encapsulation, meaning that they cannot easily model true ADTs like our Set or $Graph$ examples. We only allow pattern matching on sealed classes and retain encapsulation on (open) classes. Thirdly, if they want to model trully sealed algebraic datatypes they still need to resort to a separate **data** declaration. Furthermore, they do not allow GADTs (only Haskell 98 style datatypes). Finally, they do not have a formalization of their system.

Object-oriented programming and Scala The system that we propose can be considered as a simple purely functional object-oriented programming language (without subtyping). Although mainstream object-oriented languages are imperative, there is a rich theoretical literature on functional variants. Our encoding of objects is essentially a simplification of the well-known encoding based on recursive records such as the one used by Bruce (1993). Despite the simplicity of our encoding, the system we propose is practical: we can basically define, with the same convenience, the programs that we would define in Haskell and we can more conveniently define other (more object-oriented style) programs.

Much of the inspiration for the syntax of our class/object system comes from Scala (Odersky 2006), which is an impure functional object-oriented language. Our classes are akin Scala's notion of **trait** (Schärli et al. 2003); our objects stand in between Scala's notion of **object** and **class**; implicit objects are inspired by implicit objects/values; and, finally, we also borrow the concept of a **sealed class**. While on the surface Scala and our approach have some resemblances, the fact is that the core calculus (see Cremet et al. (2006) for the Featherweight Scala calculus) and the way many of the language mechanisms work are very different.

Modules In their work on named instances Kahl and Scheffczyk (2001) note that type classes in their system can be seen as a form of lightweight modules. Shields and Peyton Jones (2002) propose first-class modules after observing that there is a big overlap between Haskell's module language and its (very expressive) core language and show how many of the features of ML-style modules can be encoded in their system. Wehr (2005) introduces type-preserving translations from modules into type-classes and vice-versa and discusses the differences between the two mechanisms. Dreyer et al. (2007) propose an explicitly-typed module language where a type class programming style is supported as a particular usage mode of modules.

Similarly to named instances, our work can also be considered as a form of lightweight modules without supporting for nested modules and opaque types. It is not a surprise that our system has limitations when perceived as a module mechanism, but it was not the goal of this paper to unify type classes and modules. Nonetheless, unlike named instances and many of the module proposals, our "lightweight modules" are first class. Moreover, module systems like the one presented by Dreyer et al. (2007) tend to require sophisticated machinery that we do not need for our purposes. It would be interesting to investigate a single mechanism that unifies modules, GADTs and type classes in the future.

6. Conclusion and Future Work

We presented a design which unifies type classes and datatypes where the main constructs are **class** and **object**. Classes allow the definition of new (record) types and we can also define (closed) GADTs by using *sealed classes*. Objects generalize type class instances and value constructors. In future work, we intend to implement the system and to extend our system to support subclasses.

Acknowledgements

We would like to thank Jeremy Gibbons, Meng Wang and Philippa Cowderoy for several insightful discussions.

References

Kim B. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 1993. ACM.

Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13, New York, NY, USA, 2005a. ACM.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*, pages 241–253, New York, NY, USA, 2005b. ACM Press.

Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for scala type checking. In *Proc. MFCS*, Springer LNCS, September 2006.

Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998.

Atze Dijkstra and S. Doaitse Swierstra. Making implicit parameters explicit. Technical Report UU-CS-2005-032, Department of Information and Computing Sciences, Utrecht University, 2005.

Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. Modular type classes. *SIGPLAN Not.*, 42(1): 63–70, 2007.

C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18 (2):109–138, 1996.

John Hughes and Jan Sparud. Haskell++: An object-oriented extension of haskell. In *Haskell Workshop*, 1995.

Mark P. Jones. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer-Verlag.

Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 52–61, New York, NY, USA, 1993. ACM.

Wolfram Kahl and Jan Scheffczyk. Named instances for haskell type classes. In Ralf Hinze, editor, *Proc. Haskell Workshop 2001*, volume 59, 2001.

Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. 2005.

Donald Knuth. Semantics of context-free grammars. *Mathematical Systems Theory*, 2:127–145, 1968.

Jeffrey R. Lewis, Mark Shields, John Launchbury, and Erik Meijer. Implicit parameters: Dynamic scoping with static types. In *Symposium on Principles of Programming Languages*, pages 108–118, 2000.

Erik Meijer and Koen Claessen. The Design and Implementation of Mondrian. In *Haskell Workshop*. ACM, June 1997.

Johan Nordlander. *Reactive Objects and Functional Programming*. PhD thesis, Chalmers University of Technology, 1999.

Martin Odersky. An Overview of the Scala programming language (second edition). Technical Report IC/2006/001, EPFL Lausanne, Switzerland, 2006.

Bruno C.d.S Oliveira and Jeremy Gibbons. TypeCase: a design pattern for type-indexed functions. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, pages 98–109, New York, NY, USA, 2005. ACM Press.

Simon Peyton Jones. Bulk types with class. In Phil Trinder, editor, *Electronic proceedings of the 1996 Glasgow Functional Programming Workshop*, 1996.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, New York, NY, USA, 2006. ACM Press.

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *ECOOP 2003 – Object-Oriented Programming*, volume 2743, July 2003.

Tim Sheard. Putting curry-howard to work. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 74–85, New York, NY, USA, 2005. ACM.

Mark B. Shields and Simon Peyton Jones. First class modules for Haskell. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, pages 28–40, January 2002.

Christopher Strachey. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, August 1967. Reprinted in *Higher-Order and Symbolic Computation*, 13(1/2), pp. 1–49, 2000.

Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*. ACM, 2007.

Philip Wadler. The expression problem. Java Genericity Mailing list, November 1998.

Stefan Wehr. ML modules and Haskell type classes: A constructive comparison. Master's thesis, Albert-Ludwigs-Universität, Freiburg, Germany, November 2005.